Carrera de robots

Práctica 2 - Visión Artificial y Robótica
4º curso de Ingeniería Informática, UA

Clara González Sánchez

Noelia Luna Barahona

Índice

1. Introducción

El objetivo principal de esta segunda práctica es que el robot complete de forma totalmente autónoma una vuelta al circuito de Fórmula 1. Además, se propone la detección de objetos para realizar la vuelta esquivando los obstáculos que se encuentran en el camino.

Para realizar este proyecto se utilizará el lenguaje de programación Python y la plataforma Unibotics. La meta es lograr que el robot conduzca de manera independiente en el ejercicio “Obstacle Avoidance”, combinando la navegación autónoma y la detección de obstáculos.

Debido a las limitaciones de espacio, a Moodle hemos subido únicamente la memoria, el código y el modelo que mejor nos ha funcionado para resolver el ejercicio. Todo el código desarrollado, junto con los datos, datasets, resultados, modelos y scalers utilizados se encuentran disponibles en el siguiente enlace:

https://drive.google.com/file/d/12FV2nf_3ExNp565rf4jddefN-8ZNnqcP/view?usp=sharing

2. Propuesta

El enfoque propuesto para el desarrollo de la práctica consiste en las siguientes etapas:

a. Creación de un dataset para realizar aprendizaje supervisado.
b. Entrenamiento de modelos de Inteligencia Artificial.
c. Prueba del modelo entrenado en el circuito.

a. Creación del dataset

El primer paso para la creación del dataset es conseguir que el coche de vueltas al circuito de manera autónoma. Este paso se complicó más de lo que teníamos previsto, ya que la plataforma de Unibotics tiene muchas funcionalidades restringidas con tal de proveer un entorno de aprendizaje sencillo, con lo que no podíamos utilizar un método de entrada por teclado para dirigir a nuestro coche. Para resolver esta parte acabamos utilizando el algoritmo Virtual Force Field (VFF) para la evitación de obstáculos. Este algoritmo permite al robot detectar y esquivar obstáculos de manera eficiente mientras el coche circula. Para la implementación del mismo se ha partido de la implementación propuesta por los desarrolladores de Unibotics y disponible en el repositorio de GitHub [1].

Este código es prácticamente ya funcional, pero se han hecho pequeños ajustes para utilizar los métodos disponibles en el ejercicio concreto. Además, hemos utilizado las librerías Pandas y Numpy para almacenar las lecturas del sensor LiDAR, las velocidades lineal y angular en un dataframe, y posteriormente guardarlas en un archivo. Dado que el robot no para cuando finaliza una vuelta al circuito, el volcado de las lecturas a un archivo se hace cada 1.000 iteraciones del código.

Los datos recogidos dando vueltas al circuito utilizando el algoritmo VFF se han extraído del contenedor Docker donde se desarrolla el ejercicio (comando cp idContenedor:/rutaEnContenedor /rutaLocal). A continuación, hemos procesado estos datos

para eliminar valores pérdidos y corregir lecturas con valor de infinito. Hemos decidido asignar a estos valores una medida de 20, porque tras realizar un estudio descriptivo hemos visto que la máxima medida proporcionada por los LiDAR está en el rango [9, 10).

Hemos creado dos datasets, uno inicial con aproximadamente 4.000 datos recogidos al dar una vuelta, y otro más grande con los datos recogidos al dar 3 vueltas al circuito (en torno a 12.000 muestras).

Dado que las capturas de datos las hemos tomado manteniendo una velocidad lineal constante y modificando la velocidad angular para esquivar los obstáculos, hemos decidido modelar el ejercicio como un problema de regresión, de manera que utilizamos las lecturas del LiDAR como entradas de nuestros modelos y la variable de salida a predecir es la velocidad angular. Esta forma de modelar el problema nos ha permitido generar el dataset y etiquetarlo de manera automática, sin necesidad de hacer el etiquetado de forma manual, lo cuál hubiese resultado muy costoso temporalmente. Esta parte del ejercicio se realiza en el programa *obstacle_avoidanceVFF.py*.

b. Entrenamiento de modelos de IA

Como hemos mencionado anteriormente, la solución que proponemos se basa en aprendizaje supervisado. Así, hemos elegido los algoritmos de Random Forest y Neural Network.

Random Forest es un algoritmo de aprendizaje supervisado que utiliza múltiples árboles de decisión. Cada árbol se entrena con una muestra aleatoria del dataset, y las predicciones de todos los árboles se combinan para mejorar la precisión y evitar el sobreajuste. Hemos elegido este algoritmo porque es robusto frente a datos ruidosos, como pueden ser las lecturas del LiDAR, y tiene una alta capacidad de generalización, lo que permitirá realizar predicciones aunque los obstáculos cambien ligeramente de posición en el circuito. Este algoritmo es particularmente útil para problemas donde la relación entre las características y la variable de salida no es lineal ni simple. Para este algoritmo hemos utilizado la implementación de Scikit-Learn denominada RandomForestRegressor [2].

Por otro lado, las redes neuronales son modelos de aprendizaje profundo que consisten en múltiples capas de nodos (neuronas). Cada capa transforma los datos de entrada a través de funciones de activación no lineales, permitiendo al modelo aprender y generalizar patrones complejos. Hemos decidido utilizar este modelo porque las redes neuronales son capaces de capturar relaciones intrincadas entre los datos, lo que las convierte en una herramienta altamente eficaz para tareas que implican el reconocimiento de patrones complejos. La implementación que hemos decidido utilizar es una red neuronal creada mediante Keras [3].

c. Testeo de los modelos

Una vez entrenados los modelos, los hemos probado en el circuito propuesto para evaluar su rendimiento y capacidad de conducción autónoma mediante un programa desarrollado en python (*autonomous_driver.py)*. Para que funcione correctamente este programa hay

que realizar la copia de los modelos guardados al contenedor docker (comando *docker cp /rutaLocal id_contenedor:/rutaArchivosEnDocker/*) e instalar las librerías *joblib* y *scikit-learn*.

Este programa se inicia con la carga del modelo y a continuación realiza el bucle principal del programa. En este bucle se obtiene la información del sensor LiDAR, se transforma a un array de Numpy y se corrigen los posibles valores infinitos que pueda haber en las lecturas. A continuación, se transforma en un vector 2D y se utiliza para que el modelo realice predicciones sobre la velocidad angular. El resultado de esa predicción se utiliza para establecer la velocidad angular del coche en ese momento. Como ya hemos mencionado, la velocidad lineal se mantiene constante en todo momento de igual manera que con el algoritmo VFF.

3. Experimentación

Para esta práctica, hemos realizado una partición inicial del conjunto de datos en conjuntos de entrenamiento y test, utilizando una proporción del 80-20.

Métricas

Las métricas empleadas para evaluar el rendimiento de los modelos han sido MAE, MSE y R^2. Estas métricas nos permiten medir la precisión de las predicciones y la capacidad de generalización de los modelos.

El Error Absoluto Medio (MAE) calcula el promedio de las diferencias absolutas entre las predicciones del modelo y los valores reales. Por otro lado, el Error Cuadrático Medio (MSE) calcula el promedio de los cuadrados de las diferencias entre las predicciones y los valores reales. Ambas métricas ofrecen una medida de la precisión del modelo, pero el MSE penaliza de manera más significativa los errores grandes debido a la operación de elevar al cuadrado.

El coeficiente de determinación, comúnmente conocido como R cuadrado (R^2), es una métrica estadística utilizada en regresión para evaluar cómo de bien se ajusta el modelo de regresión a los datos observados. Es decir, es una medida de la proporción de la variación en la variable dependiente que es explicada por el modelo de regresión. Esta métrica toma valores entre 0 y 1. Un valor de R cuadrado cercano a 1 indica que una gran proporción de la variación en la variable dependiente es explicada por el modelo, lo que sugiere un buen ajuste. Por otro lado, un valor de R cuadrado cercano a 0 indica que el modelo no explica mucha variación en la variable dependiente y puede indicar un mal ajuste.

Experimento 1 - Random Forest, dataset pequeño

En la primera aproximación, hemos utilizado el algoritmo Random Forest con valores por defecto y un conjunto de 4.000 datos de entrenamiento aproximadamente. Esta configuración inicial permite establecer una línea base para evaluar el rendimiento del modelo y comprender cómo maneja una cantidad moderada de datos. Las métricas obtenidas han sido:

MAE	MSE	R^2
0.0724	0.0377	0.9687

De acuerdo con estas métricas, el ajuste del modelo es bastante bueno. Con esta implementación el robot consigue dar una vuelta completa al circuito sin colisionar con los obstáculos en un tiempo en torno a 3 minutos. No obstante, en ocasiones falla y sí choca contra los obstáculos o se sale del circuito.

Experimento 2 - Random Forest, dataset grande

En el segundo experimento, hemos incrementado el conjunto de datos de entrenamiento a 12.000 datos, manteniendo los valores por defecto del algoritmo Random Forest para ver si mejoraba el rendimiento del modelo. Las métricas obtenidas fueron:

MAE	MSE	R^2
0.0696	0.0308	0.9754

Las métricas son ligeramente superiores a las del experimento anterior, con un ajuste muy bueno del modelo. Con esta implementación el coche consigue dar una vuelta completa al circuito sin colisionar con los obstáculos en un tiempo en torno a 3 minutos. No obstante, similar a como ocurre en el caso anterior, alguna vez se choca contra alguno de los obstáculos, aunque en menor medida que el modelo entrenado con el dataset pequeño.

https://youtu.be/235Rqf3pIlE

Experimento 3 - Ajuste de Random Forest

Para encontrar la mejor configuración para el modelo de Random Forest [2] se ha utilizado GridSearchCV [4], que realiza una validación cruzada utilizando combinaciones de los parámetros de configuración especificados. Hemos empleado una K de 5 para asegurar una evaluación completa y robusta de los modelos.

La elección de la mejor configuración de cada modelo se ha basado en la métrica *neg_mean_squared_error*, que es el valor negativo de MSE. Hemos utilizado esta métrica porque las funciones de scoring en GridSearchCV están diseñadas para maximizar la puntuación. Al emplear el valor negativo de MSE, una puntuación más alta (menos negativa) indica un mejor modelo. A continuación se detallan los parámetros de configuración utilizados y sus valores.

- *'n_estimators'*: [100, 200, 300, 500],
- *'max_depth'*: [10, 20, 30, 40],
- *'max_features'*: ['auto', 'sqrt', 'log2'],
- *'criterion'*: ['squared_error', 'absolute_error']

El parámetro *max_depth* especifica la profundidad máxima del árbol de decisión. Controla la cantidad de divisiones que se permiten en el árbol antes de que se alcance la profundidad máxima. Valores más altos pueden conducir a árboles más complejos y propensos al sobreajuste, mientras que valores más bajos pueden resultar en árboles más simples y propensos al subajuste. Se han utilizado los valores 10, 20, 30, 40, para explorar una amplia gama de profundidades posibles.

El parámetro *criterion* especifica la función utilizada para medir la calidad de una división en el árbol. Las opciones empleadas han sido el error cuadrático medio (MSE) y el error absoluto medio (MAE).

El parámetro *max_features* controla el número máximo de características que se consideran al buscar la mejor división en cada nodo del árbol. Se han empleado tres opciones:

- 'auto': Considera todas las características para la división en cada árbol.
- 'sqrt': Considera la raíz cuadrada del número total de características.
- 'log2': Considera el logaritmo base 2 del número total de características.

El parámetro *n_estimators* especifica el número de árboles en el bosque. Cuantos más árboles se incluyan, más robusto será el modelo y menos probable será que sufra de sobreajuste. Los valores utilizados han sido 100, 200, 300 y 500, para así determinar el número óptimo de árboles para el bosque.

Este experimento lo hemos realizado con aproximadamente 4.000 datos de entrada, ya que requiere mucho tiempo de computación (225 min y 36,8 segundos, específicamente).

Los parámetros proporcionados los hemos utilizado para entrenar modelos en ambos datasets. Las métricas obtenidas para el dataset pequeño han sido:

MAE	MSE	R^2
0.0831	0.0438	0.9636

Mientras que para el dataset grande hemos obtenido:

MAE	MSE	R^2
0.0805	0.0290	0.9769

De nuevo, en los dos casos se consigue un ajuste bastante bueno y ambos modelos logran una conducción autónoma correcta, pero en ocasiones siguen colisionando con algún obstáculo del camino. En el siguiente enlace se puede visualizar el vídeo del RF ajustado con el dataset grande:

https://youtu.be/0kDFnJIQGVY

Experimento 4 - Random Forest con StandardScaler

Otro de los experimentos realizados ha consistido en aplicar estandarización a los datos. La estandarización es un proceso en el que los datos se transforman de manera que tengan una media de cero y una desviación estándar de uno. Este proceso es útil para hacer que las características de los datos estén en la misma escala, lo que puede mejorar el rendimiento de ciertos modelos de aprendizaje automático, especialmente aquellos sensibles a las diferencias de escala entre las características.

La razón de aplicar esta técnica es que puede mejorar la convergencia del modelo y hacer que sea más fácil para el algoritmo de optimización encontrar el mínimo global de la función de pérdida. Las métricas obtenidas han sido:

MAE	MSE	R^2
0.0693	0.0309	0.9754

Si bien las métricas demuestran un buen ajuste del modelo, al utilizarlo para la conducción autónoma por el circuito los resultados no han sido satisfactorios. La conducción es más suave, sin tantas correcciones pero el coche ha colisionado con el primer o segundo obstáculos, o bien se ha salido de la pista. En el siguiente vídeo se puede ver una demostración de lo que ocurre:

https://youtu.be/VoyNM831yeQ

Asociamos el que no vaya bien a que las mediciones de los sensores son muy precisas y con un rango bastante amplio, por lo que que al intentar escalar estos valores puede ocurrir que estemos perdiendo información y el modelo deje de ser capaz de generalizar.

Experimento 5 - RF con MinMaxScaler

Otro experimento llevado a cabo ha implicado la normalización de los datos utilizando el método Min-Max Scaling, también conocido como MinMaxScaler. La normalización Min-Max es un proceso en el que los valores de las características se transforman de manera que estén dentro de un rango específico, como por ejemplo [0, 1].

La razón de utilizar esta técnica es la misma que para la estandarización. Las métricas obtenidas han sido:

MAE	MSE	R^2
0.0700	0.0308	0.9754

De nuevo, las métricas muestran un buen ajuste del modelo, pero al ejecutar la simulación ha ocurrido lo mismo que con la estandarización de datos.

https://youtu.be/aszsPWLGM8Y

Asociamos de nuevo el fracaso de esta técnica al mismo motivo del apartado anterior. Sin embargo, no es una pérdida de tiempo ya que toda experimentación es un avance en decidir qué puede funcionar y qué no.

Experimento 6 - Red neuronal

A continuación, decidimos probar una red neuronal sencilla para regresión con la siguiente arquitectura y funciones de activación:

```
model = Sequential([
            layers.Input(shape=(X_train.shape[1],)),
            layers.Dense(128, activation='relu'),
            layers.Dense(64, activation='relu'),
            layers.Dense(32, activation='relu'),
            layers.Dense(1, activation='linear'),
        ])
```

El optimizador elegido ha sido Adam y el número de épocas de entrenamiento ha sido 100. Las métricas obtenidas han sido:

MAE	MSE	R^2
0.1660	0.2291	0.8678

Se puede observar que el ajuste conseguido no alcanza los mismos valores que con el Random Forest. Con esta implementación no se ha logrado una vuelta completa al circuito, ya que en algún momento del mismo acaba colisionando con uno de los obstáculos.

https://youtu.be/fqrX9Tqz68k

El aumento de las épocas de entrenamiento de la red neuronal probablemente mejore los resultados obtenidos con este algoritmo. No obstante, el aumento de 100 a 200 épocas, sólo mejoró unas milésimas el resultado, aumentando bastante el tiempo de computación necesario.

Experimento 6 - Mezclado de datos antes de generar el dataset

Finalmente, dado que los datos se recogen de manera secuencial y en el orden en el que se realiza el circuito, hemos decidido realizar un *shuffle* de los mismos, alterando así el orden en el que aparecen en el dataset. Esta fase se ha realizado antes de separar los conjuntos X e Y, y de nuevo hemos dividido en entrenamiento y test en una proporción de 80-20.

Hemos utilizado estos datos para entrenar de nuevo tanto RF como NN. En el caso de RF, se han resuelto las colisiones ocasionales con obstáculos, logrando que el coche de 3 vueltas completas sin chocar. En el siguiente link se incluye la grabación de una vuelta, en un tiempo de 3 minutos y 13 segundos.

https://youtu.be/Nv8ZMe7MVn4

Por otro lado, el modelo resultante de la red neuronal ha seguido colisionando con coches y obstáculos. En el siguiente enlace se puede visualizar el resultado:

https://youtu.be/HcvGSv9yq70

4. Conclusiones

En nuestro caso, los modelos realizados mediante Random Forest han sido consistentemente mejores que los experimentos que hemos realizado con Redes Neuronales, por lo que hemos profundizado mucho más en este primer tipo. Concretamente, hemos logrado los mejores resultados al usar nuestro dataset más grande y mezclándolos para realizar el entrenamiento.

Por otro lado, las técnicas de normalización y estandarización de datos han empeorado el rendimiento del modelo, contrario a lo que hubiésemos esperado.

Una gran limitación que hemos tenido ha sido que nuestros modelos resultantes tuvieran que hacer predicciones en tiempo real, lo cuál nos ha limitado las técnicas que podíamos aplicar. No obstante, el resultado final ha sido un modelo que consigue su cometido, dando vueltas al circuito sin chocar con ninguno de los obstáculos en su camino.

Por último queremos resaltar que para conseguir una conducción autónoma real se hace necesario el entrenamiento en otros circuitos y situaciones.

5. Referencias

[1] S. Mahna. “VFF- Robotics-Playground”. GitHub. Accedido el 18 de mayo de 2024. [En línea]. Disponible: https://github.com/SakshayMahna/Robotics-Playground/blob/main/f1_ws/src/local_navigation/scripts/vff.py

[2] Scikit-Learn. “RandomForestRegressor”. Scikit-learn. Accedido el 18 de mayo de 2024. [En línea]. Disponible: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html

[3] F. Chollet. “Keras”. [En línea]. Disponible: https://keras.io

[4] Scikit-Learn. “GridSearchCV”. scikit-learn. Accedido el 18 de mayo de 2024. [En línea]. Disponible: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html