Universidad Politécnica de Madrid
Escuela Técnica Superior de Ingenieros de Telecomunicación

APLICACIÓN DE TENSORFLOW EN DEEP LEARNING

TRABAJO FIN DE MÁSTER

Alfredo Valle Barrio

2017-2018

Universidad Politécnica de Madrid
Escuela Técnica Superior de Ingenieros de Telecomunicación

Máster Universitario en
Ingeniería de Redes y Servicios Telemáticos

TRABAJO FIN DE MÁSTER

APLICACIÓN DE TENSORFLOW EN DEEP LEARNING

Autor
Alfredo Valle Barrio

Director
José Eugenio Naranjo Hernández
Ponente
Joaquín Salvachúa Rodríguez

Departamento de Ingeniería de Sistemas Telemáticos

2017-2018

Resumen

Palabras clave: Red neuronal convolucional, rendimiento, odometría visual, LIDAR, Tensorflow, vehículo autónomo, INSIA.

Los vehículos autónomos son ya prácticamente una realidad y sus primeras implementaciones comienzan a aparecer. Y con ellas los primeros problemas.

En este proyecto se ha querido abordar uno de ellos, concretamente el problema de odometría. Este problema es especialmente importante en un vehículo autónomo debido a que para su posicionamiento este, debe fiarse únicamente de los sensores a los que tiene acceso, los cuales pueden tener errores importantes que lleven a consecuencias muy graves. Por ello, la motivación de este trabajo es proporcionar al vehículo una nueva manera de auto localizarse con el fin de minimizar estos errores y con ello sus consecuencias.

En este proyecto, se ha tratado de desarrollar algoritmos que realicen una odometría visual a partir de los datos de un sensor LIDAR mediante redes neuronales. Aunque también, se ha tratado enfocar el trabajo en el estudio del funcionamiento y rendimiento de las redes neuronales implementadas sobre Tensorflow.

Para realizar este proyecto, este se ha dividido en varias fases. Una primera fase de familiarización con las redes neuronales, redes neuronales convolucionales y Tensorflow. A continuación, se han desarrollado algoritmos relativamente sencillos de odometría visual, con los que se han realizado estudios del comportamiento y rendimiento de Tensorflow y del Hardware, con distintas configuraciones del mismo.

Por último, una vez realizada la experimentación con el Hardware y habiendo encontrado las configuraciones más adecuadas, se ha estudiado más detenidamente el rendimiento del Software, para lo cual se han desarrollado mucho más las arquitecturas de las redes neuronales utilizadas tratando de hacerlas más precisas. Y utilizando técnicas que no se conocían anteriormente.

Para finalizar, de toda esta experimentación se han extraído una serie de conclusiones, concernientes tanto a la arquitectura Hardware optima a utilizar en este tipo de sistemas, como a las técnicas de implementación de redes neuronales convolucionales mediante Tensorflow que proporcionan una fiabilidad más alta.

Abstract

Keywords: Convolutional neuronal network, performance, visual odometry, LIDAR, Tensorflow autonomous vehicle, INSIA.

Autonomous vehicles are practically a reality and their first implementations begin to appear. And with them the first problems.

In this project we wanted to address one of them, specifically the problem of odometry. This problem is especially important in an autonomous vehicle because for its positioning, it must rely only on the sensors to which it has access, which can have important errors that lead to very serious consequences. Therefore, the motivation of this work is to provide the vehicle with a new way of locating itself in order to minimize these errors and thereby its consequences.

In this project, we have tried to develop algorithms that perform a visual odometry from the data of a LIDAR sensor through neural networks. Although also, it has been tried to focus the work in the study of the functioning and performance of the neural networks implemented on Tensorflow.

To perform this project, it has been divided into several phases. A first phase of familiarization with neural networks, convolutional neural networks and Tensorflow. Then, relatively simple algorithms of visual odometry have been developed, with which studies of the behaviour and performance of Tensorflow and the Hardware have been carried out, with different configurations of it.

Finally, once the experimentation with the Hardware has been done and having found the most appropriate configurations, the performance of the Software has been studied more carefully, for which the architectures of the neural networks used have been developed much more trying to make them more precise. And using techniques that were not previously known.

To conclude, a series of conclusions have been drawn from all this experimentation, concerning both the optimal hardware architecture to be used in this type of systems, and the techniques for implementing convolutional neural networks using Tensorflow that provide higher reliability.

Índice general

Resumen i
Abstract iii
Índice general v
Índice de figuras ix
Siglas xi
1 Introducción 13
2 Estado del Arte 15
2.1 Historia de los vehículos autónomos 15
2.1.1 La patente que cambio el mundo 15
2.1.2 Torpedo aéreo de Sperry (Primer piloto automático) 16
2.1.3 UAV (Vehículo aéreo no tripulado) 16
2.1.4 Seguridad pasiva en los vehículos 16
2.1.5 Seguridad activa en los vehículos 16
2.1.6 Lanzamiento del primer vehículo autónomo 17
2.2 Tecnologías 17
2.2.1 LIDAR 18
2.2.2 Cámara 20
2.2.3 Cámara estéreo 21
2.2.4 Radar 22
2.2.5 GPS 22
2.2.6 IMU 23
2.2.7 Ultrasonido 24
2.3 Procesamiento de la información 24
2.3.1 Fusión de sensores 24
2.3.2 Inferencia Bayesiana 25
2.3.3 Auto localización (SLAM) 25
2.3.4 Detección y evaluación de obstáculos 26
2.3.5 Planificación de trayectorias 27

2.3.6 Evasión de obstáculos ... 29
2.3.7 Toma de decisiones ... 29
2.4 Conducción autónoma ... 30
2.4.1 Niveles de clasificación de coches autónomos ... 31
Nivel 0 ... 31
Nivel 1 ... 31
Nivel 2 ... 31
Nivel 3 ... 32
Nivel 4 ... 32
Nivel 5 ... 32
2.4.2 Sistemas autónomos actuales ... 32
Vehículos semiautónomos actuales ... 32
2.4.3 Previsión del desarrollo de los vehículos autónomos ... 37
2.5 Historia de la inteligencia artificial ... 38
2.5.1 Termino de inteligencia artificial ... 38
2.5.2 Sistemas expertos ... 38
2.5.3 Lenguaje Wave ... 39
2.5.4 Agentes inteligentes ... 39
2.5.5 Robots ... 39
2.5.6 Inteligencia artificial de Google ... 40
3 Metodología ... 41
3.1 Lenguajes de programación empleados ... 41
3.1.1 Python ... 41
3.2 Librerías especializadas ... 41
3.2.1 Tensorflow ... 41
3.3 Entornos de desarrollo ... 42
3.3.1 PyCharm ... 42
3.4 Git ... 42
3.5 INSIA ... 43
4 Trabajo Desarrollado ... 45
4.1 Instalación ... 45

4.1.1 CUDA ... 45
4.1.2 cuDNN ... 45
4.1.3 Preinstalación ... 46
Limpieza de antiguas instalaciones ... 46
4.1.4 Instalación CUDA ... 46
Preinstalación ... 46
Instalación ... 47
Verificación ... 49
4.1.5 Instalación de cuDNN ... 49
Descarga de cuDNN ... 49
Instalación ... 50
Verificación ... 50
Observaciones ... 51
4.1.6 Montar maquina Azure ... 51
Pasos para crear la máquina virtual ... 51
Creación de máquina virtual Azure desde plantilla ... 54
4.2 Software ... 57
4.2.1 Software para la maquina local ... 57
4.2.2 Software para Azure ... 62
4.3 Estudio de rendimiento Hardware ... 69
4.3.1 Hardware utilizado ... 69
Hardware Local ... 69
Hardware Azure ... 69
4.3.2 Pruebas en la maquina local ... 70
Escenarios hardware ... 70
4.3.3 Escenario 1(CPU) ... 70
4.3.4 Escenario 2(2 graficas) ... 72
4.3.5 Escenarios 3 y 4 ... 74
4.3.6 Pruebas en la maquina Azure ... 78
5 Conclusiones ... 83
6 Líneas de trabajo futuro ... 85

7 Bibliografía 87

Índice de figuras

Figura 1: Línea de tiempo 15
Figura 2: Teleautomaton de Tesla 15
Figura 3: UAV 16
Figura 4: Vehículo autónomo de Google 17
Figura 5: Estructura de sensores en un vehículo autónomo 18
Figura 6: Sensores LIDAR 19
Figura 7: Imagen sensor LIDAR 19
Figura 8: Cámara Mobileye 20
Figura 9: Visión de una cámara Mobileye 20
Figura 10: Cámara estéreo 21
Figura 11: Visión de una cámara estéreo 21
Figura 12: Radar 22
Figura 13: GPS 22
Figura 14: Unidad de movimiento inercial 23
Figura 15: Sensor de Ultrasonidos 24
Figura 16: Ejemplo de planificación global de trayectorias 28
Figura 17: Aproximación de trayectoria en una curva mediante curvas de Bézier 28
Figura 18: Aproximación de trayectoria en curvas consecutivas mediante curvas de Bézier (6) 29
Figura 19: Evasión de obstáculos mediante campos potenciales (7) 29
Figura 20: Aproximación de adelantamiento mediante lógica difusa (8) 29
Figura 21: Niveles de autonomía de un vehículo autónomo 31
Figura 22: Coche autónomo Mercedes Benz 32
Figura 23: Vehículo autónomo Waymo 33
Figura 24Visión del entorno de los vehículos Waymo 34
Figura 25: Coche autónomo Uber 34
Figura 26: Prototipo de vehículo autónomo Uber - Volvo 35
Figura 27: Coche autónomo tesla 35
Figura 28: Percepción del entorno de un vehículo Tesla 36
Figura 29: Coche autónomo Baidu 36
Figura 30: Previsión de desarrollo de los vehículos autónomos 37
Figura 31: Asimo 40
Figura 32: Interfaz PyCharm 42
Figura 33: Panorámica de las instalaciones del INSIA 44

Figura 34: Selección de plataforma para la descarga de CUDA 47
Figura 35: Creación VM Azure(Básico) 52
Figura 36: Maquinas disponibles 53
Figura 37: Editor de plantillas 57
Figura 38: Grafo red neuronal local 62
Figura 39: Grafo código Azure 67
Figura 40: Función de perdida Azure 68
Figura 41: Potencia consumida ambas gráficas 71
Figura 42: Memoria utilizada ambas gráficas 71
Figura 43: Porcentaje de utilización ambas gráficas 71
Figura 44: Potencia consumida ambas gráficas 73
Figura 45: Memoria utilizada ambas gráficas 73
Figura 46: Porcentaje de utilización ambas gráficas 73
Figura 47: Potencia consumida ambas gráficas 75
Figura 48: Memoria utilizada ambas gráficas 75
Figura 49: Porcentaje de utilización ambas gráficas 76
Figura 50: Potencia consumida ambas gráficas 76
Figura 51: Memoria utilizada ambas gráficas 77
Figura 52: Porcentaje de utilización ambas gráficas 77
Figura 53: Potencia consumida 78
Figura 54: Memoria utilizada 79
Figura 55: Utilización 79
Figura 56: Potencia consumida 80
Figura 57: Memoria utilizada 81
Figura 58: Utilización 81

Siglas

LIDAR: Laser Imaging Detection and Ranging

CPU: Central Process Unit

GPU: Graphics Processing Unit

TPU: Tensor Processing Unit

UAV: Unmanned Aerial Vehicle

IEEE: Institute of Electrical and Electronics Engineers

INSIA: Instituto universitario de investigación del automóvil

ETSII: Escuela Técnica Superior de Ingenieros Industriales

ETSIT: Escuela Técnica Superior de Ingenieros de Telecomunicación

NN: Neuronal Network

CUDA: Compute Unified Device Architecture

cuDNN: CUDA Deep Neuronal Network

1 Introducción

En la actualidad, está habiendo mucha controversia en materia de vehículos autónomos, debido a recientes incidentes en los que ha acabado herida alguna persona. Todos los grupos de investigación dedicados a los vehículos autónomos tienen muy presente el compromiso de seguridad y fiabilidad que estos deben tener. Por ello se quiere ayudar en esta materia con este proyecto. El contenido del mismo se espera que pueda llegar a ser implementado en el futuro en un vehículo autónomo.

El objetivo de este proyecto es desarrollar una red neuronal convolucional que realice una odometría visual de los datos recogidos de un láser scanner, Velodyne Lidar 16 canales (1). A su vez se ha realizado un estudio del rendimiento tanto del framework para el desarrollo de tensores, Tensorflow (2) como del Hardware utilizado, cuyos resultados son los que se exponen en este proyecto.

Para alcanzar este objetivo se ha utilizado Tensorflow como base para la implementación en Python 3.5 de una red neuronal convolucional. En cuanto al Hardware utilizado, ya que la computación de redes neuronales resulta bastante lenta sobre CPU, se ha realizado sobre sistemas con GPUs y mediante las librerías incluidas en Tensorflow (3) se ha podido ejecutar la red neuronal sobre éstas.

Los sistemas sobre los que se ha trabajado son los siguientes. En una primera fase se ha utilizado un equipo disponible en el grupo de investigación en el que se ha trabajado, que está en el INSIA (4), este servidor en un principio contaba con una tarjeta gráfica Nvidia GTX 1080 Ti, a la que fue añadida otra tarjeta gráfica cedida por envidia, una Nvidia Titan X. Ambas tarjetas soportan ejecución con CUDA y cuDNN.

Mas adelante, durante la segunda fase del proyecto se recibió una subscripción a Azure (5), cedida por Microsoft, que ha permitido virtualizar un equipo con 4 tarjetas gráficas Nvidia Tesla K80, mucho más potentes que las anteriores.

En la primera fase del proyecto, en donde las pruebas se realizaron sobre la maquina local, se estuvo investigando en las diferentes configuraciones de hardware disponibles, a través de Tensorflow. Aquí, se realizó un estudio de los comportamientos de las tarjetas gráficas, así como de la velocidad de ejecución y aprendizaje de la red neuronal.

En la segunda fase del proyecto, con los comportamientos de los dispositivos ya identificados se mejoró en gran medida la arquitectura e implementación de la red neuronal. Y a su vez el trabajo se centró el estudio en el rendimiento software, enfocándose principalmente en la velocidad de aprendizaje.

Durante el desarrollo y experimentación de este trabajo se han extraído una serie de conclusiones que se exponen al final del mismo.

En el desarrollo de esta memoria se expone el estado del arte en materia de vehículos autónomos, ya que la finalidad y motivación de este proyecto es que en un futuro lo desarrollado en este proyecto se implemente en uno de ellos. Y se expone también un breve estado del arte de la inteligencia artificial.

Para finalizar esta memoria, se exponen las líneas de investigación propuestas para la continuación de este proyecto, hacia su objetivo final.

2 Estado del Arte

En este apartado se van a abordar los cambios y avances que han sufrido las tecnologías que actualmente se utilizan en los vehículos autónomos.

2.1 Historia de los vehículos autónomos

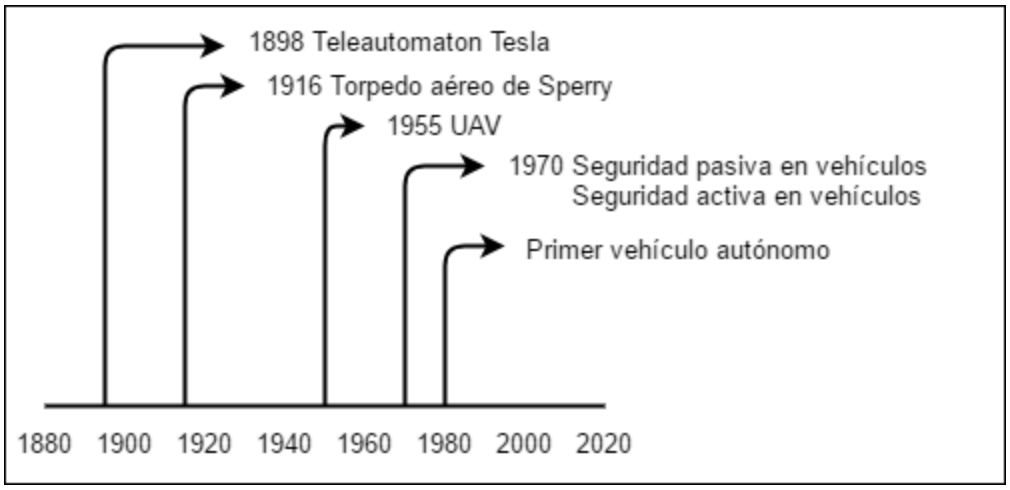

Figura 1: Línea de tiempo

2.1.1 La patente que cambio el mundo

El Teleautomaton de Nikola Tesla (1856-1943), es considerado el primer dispositivo a control remoto. Este dispositivo era un pequeño submarino controlado mediante ondas de radio, mediante las cuales se abrían o cerraban circuitos eléctricos en el interior del submarino, que suponían las instrucciones básicas.

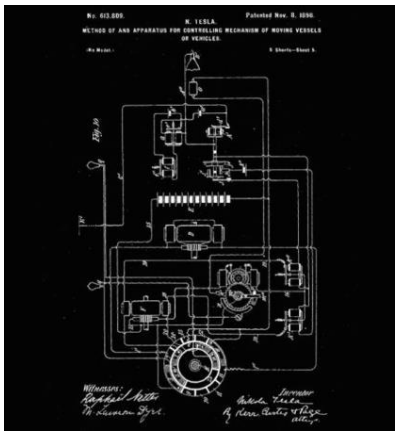

Figura 2: Teleautomaton de Tesla

En noviembre de 1898, la Oficina de Patentes concedió la patente de su dispositivo a Tesla.

2.1.2 Torpedo aéreo de Sperry (Primer piloto automático)

El torpedo aéreo de Sperry, también conocido como bomba volante, es considerado el primer piloto automático. El sistema consiste en un giroscopio estabilizador a través del cual se mantiene un avión volando en línea recta durante una determinada distancia.

La primera demostración fue realizada en 1916 para demostrar su viabilidad.

2.1.3 UAV (Vehículo aéreo no tripulado)

Los UAV o vehículos aéreos no tripulados son la evolución del torpedo aéreo de Sperry. Estos dispositivos son capaces de mantener de manera autónoma un nivel de vuelo controlado y sostenido, y propulsado por un motor.

Para que una aeronave sea considerada como UAV debe ser capaz de realizar de forma autónoma tanto el vuelo sostenido como el despegue y aterrizaje.

En la actualidad existe una gran variedad de modelos de distintas formas y tamaños. Y son operados principalmente en el ámbito militar. En la Figura 3: UAV, se puede ver la imagen de un UAV MQ-9 Reaper.

Figura 3: UAV

2.1.4 Seguridad pasiva en los vehículos

En los inicios de los años 70 se comenzó a tomar conciencia de la seguridad en el ámbito de la automoción. Durante estos años, concretamente en 1974 entró en vigor la normativa de Tráfico que obliga el uso del cinturón. A partir de ese momento cada vez son más las normativas que se han desarrollado y los cambios que han sufrido las antiguas normativas, para la protección del ocupante del vehículo.

2.1.5 Seguridad activa en los vehículos

La seguridad activa en los vehículos la componen todos aquellos elementos que aumentan la estabilidad del vehículo en movimiento y, en la medida de lo posible, evitan accidentes.

Uno de los componentes de seguridad activa más conocidos es el ABS, sistema antibloqueo de frenos. Este dispositivo reduce la distancia de frenado manteniendo la capacidad de cambiar de dirección sin bloquear las ruedas.

La primera aparición de este dispositivo fue en el año 1970 cuando Bosch desarrollo una primera versión.

2.1.6 Lanzamiento del primer vehículo autónomo

Los primeros atisbos de un vehículo autónomo fueron en el año 1980 gracias a una furgoneta guiada por visión artificial de Mercedes-Benz. Esta furgoneta fue diseñada por un equipo de la Universidad de Múnich que condujo hasta a 100 km/h en calles sin tráfico.

Gracias a este proyecto, en la que la comisión europea mostro interés e invirtió 800 millones de euros, arrancó el proyecto EUREKA Prometeus cuyo objetivo era el desarrollo de un vehículo autónomo.

En ese momento se inició la carrera para el desarrollo de un vehículo autónomo en la que estamos sumergidos en la actualidad.

Figura 4: Vehículo autónomo de Google

En la Figura 4: Vehículo autónomo de Google se puede ver la imagen del vehículo autónomo de Google, una de las empresas más avanzadas en el sector de los vehículos autónomos.

2.2 Tecnologías

En este apartado se expondrán las tecnologías que se encuentran habitualmente en un vehículo autónomo actual, así como su cometido y su potencial en el sistema. En un vehículo autónomo se pueden encontrar gran cantidad de sensores, esto se debe a que estos son el único medio de conocer el entorno que lo rodea. Por lo que de ellos depende la seguridad del sistema durante la conducción.

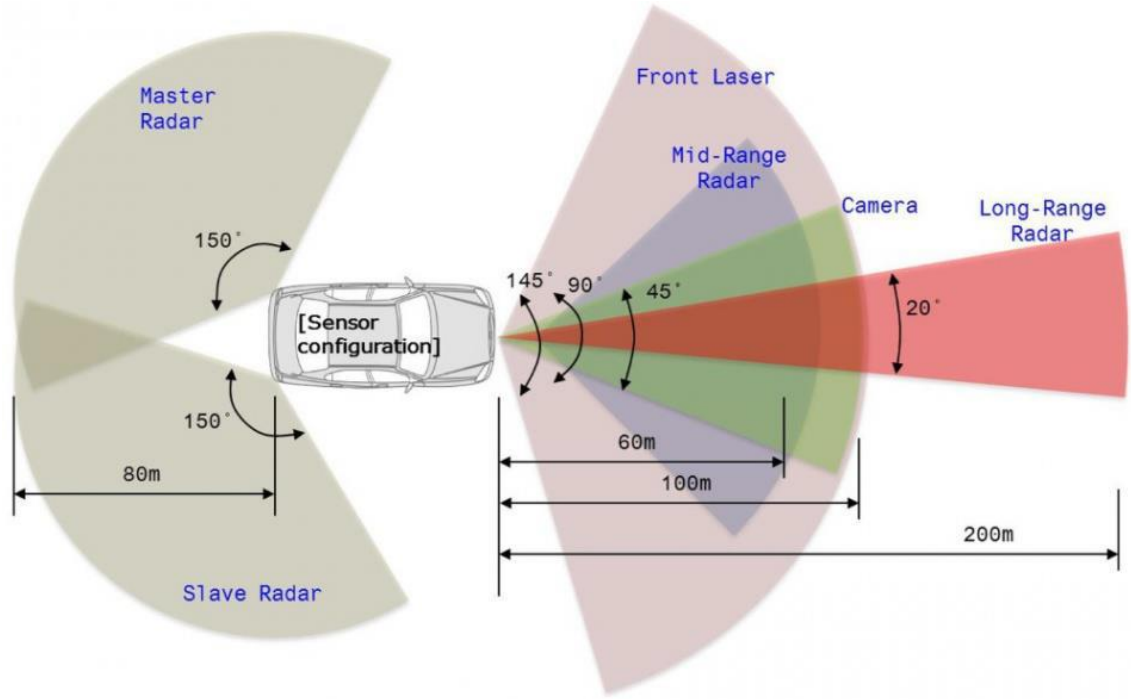

Figura 5: Estructura de sensores en un vehículo autónomo

En la Figura 5: Estructura de sensores en un vehículo autónomo, se puede observar el esquema de sensores general de un coche autónomo. En él se pueden encontrar todos los sensores que van a ser tratados en detalle a continuación.

En la lista siguiente se indica la correspondencia de cada sensor específico con cada área de interés en la imagen.

- Lidar (Front Laser)
- Cámara (Camera)
- Cámara estéreo (Camera/Mid-Range Radar)
- Radar (Long-Range Radar/Master-Slave Radar)
- GPS (Interno)
- IMU (Interno)
- Ultrasonido (Máster-Slave Radar)

2.2.1 LIDAR

Figura 6: Sensores LIDAR

LIDAR siglas en inglés de "L*aser* Imaging Detection and Ranging", es un dispositivo habitualmente conocido como láser scanner, que usa un haz pulsado de láser para medir distancias.

Un láser scanner Lidar es capaz de medir con precisión distancias de hasta 100 metros aproximadamente, pudiendo variar dependiendo del modelo.

De él se obtiene una nube de puntos tridimensionales con unas coordenadas en un sistema de coordenadas local respecto al láser en 360 grados y con una amplitud vertical que varía dependiendo del modelo. En el Velodyne Lidar de 16 canales, el más extendido, se puede encontrar una amplitud vertical de 30 grados.

Consta de uno o varios canales cada uno de los cuales es un láser que gira y realiza una medición a una frecuencia fija.

Este tipo de sensor se encuentra en prácticamente todos los coches autónomos actuales siendo muy útiles para el mapeo del entorno, así como para detección de obstáculos. También han sido desarrollados múltiples algoritmos de auto localización (SLAM) donde se ha demostrado su gran utilidad, a pesar de la gran cantidad de datos a manejar.

En la Figura 7: Imagen sensor LIDAR, se puede ver la imagen devuelta por un Lidar HDL-E de 64 canales. Se puede observar su gran resolución y distinguir de manera rápida los obstáculos del fondo. En la imagen se pueden distinguir distintos colores en la nube de puntos debido a que el Lidar también devuelve el valor de intensidad del punto en que ha rebotado, permitiendo realizar una segmentación de la nube de puntos por colores. Además, es capaz de devolver dos impactos si el objeto es relativamente transparente con lo que podemos obtener una medición tanto del objeto como del fondo.

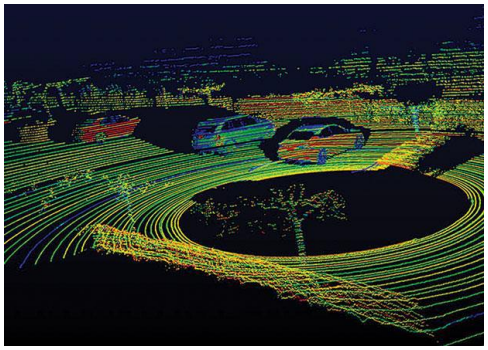

Figura 7: Imagen sensor LIDAR

2.2.2 Cámara

Figura 8: Cámara Mobileye

Se usan cámaras de propósito general para realizar trabajos de segmentación, localización y detección de señales. Pero uno de los modelos más extendidos es la cámara desarrollada por Mobileye.

La cámara Mobileye, como la que se puede ver en la Figura 8: Cámara Mobileye, es capaz detectar vehículos, bicicletas y peatones, aparte de hacer tracking de estos, pudiendo obtener su velocidad y trayectoria. También es capaz de reconocer señales de tráfico.

La cámara se conecta a la centralita del vehículo a través del CAN Bus, con lo que tiene acceso a todos los parámetros del vehículo tales como la velocidad y dirección, que le permite realizar un análisis de riesgos de cada objeto de la calzada. Todo esto es realizado mediante un procesamiento interno con lo que no se ha de realizar un post proceso de la imagen con el consiguiente ahorro en tiempo de reacción.

De ella se obtiene una evaluación del entorno, esto se puede observar en la imagen siguiente donde se pueden ver los datos obtenidos de la cámara sobre una imagen sincronizada de una cámara de propósito general.

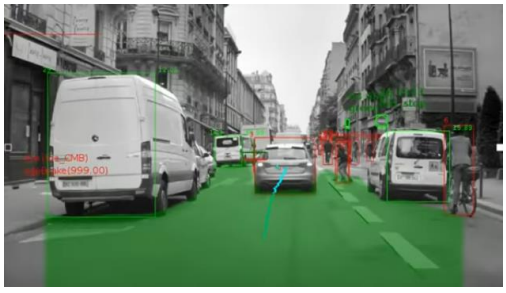

Figura 9: Visión de una cámara Mobileye

Se puede observar que reconoce varios vehículos en la carretera y su riesgo en la conducción, se puede observar también la línea azul que representa la trayectoria seguida por el vehículo de enfrente. También ha detectado la bicicleta y varios peatones.

Se puede ver que detecta los vehículos aparcados e indica que no suponen un riesgo para la conducción marcándolos en verde.

2.2.3 Cámara estéreo

Figura 10: Cámara estéreo

Una cámara estéreo es un dispositivo con dos o más lentes las cuales recoge una imagen por cada lente permitiéndole simular la visión binocular humana. Esto le permite tener una visión 3D con lo que puede hacer mediciones de distancia y profundidad.

El procesamiento de la imagen de este tipo de cámaras es similar a la de las cámaras normales como la cámara Mobileye, de la que se ha hablado anteriormente. Pero con el potencial que le da poder medir distancias mediante triangulación entre las imágenes de las dos lentes. Tal como es percibida la profundidad por los humanos.

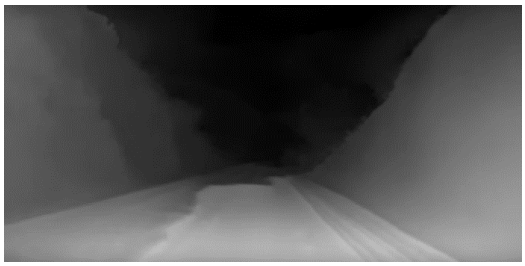

Figura 11: Visión de una cámara estéreo

En la Figura 11: Visión de una cámara , se puede ver la imagen que se obtiene de una cámara estéreo, concretamente la cámara ZED. Se puede observar que se obtiene una imagen en escala de grises, los tonos más claros representan puntos más cercanos en cambio los tonos más oscuros representan los puntos más alejados.

El verdadero potencial en el uso de esta cámara está en que se unifica la capacidad de un láser y una cámara. En cuanto al laser obtenemos la capacidad de medir distancias como se ha comentado antes, y de la cámara tenemos la resolución de puntos, mucho mayor a la de un láser scanner. Con esto también se consigue simplificar el procesamiento, ya que el procesamiento de los planos de profundidad con una cámara normal con algoritmos como el Z-Buffer es muy costoso.

2.2.4 Radar

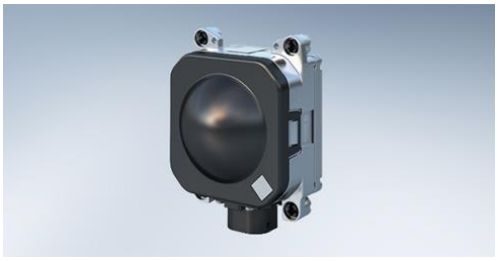

Figura 12: Radar

Los radares son otro de los sensores más usados en los coches autónomos, estos dispositivos utilizan ondas de radio para medir distancias.

A diferencia de los sensores LIDAR que trabajan con pulsos de luz, los radares trabajan en EHF, Extremely High Frequency, esta se corresponde a la banda de 30 a 300 gigahercios, definida por la Unión Internacional de Telecomunicaciones.

Estos tienen dos cometidos principales según su configuración. Pueden ser usados para medir a largas distancias, más allá del alcance de un sensor LIDAR o para detectar objetos en las proximidades. Se suelen usar para detectar obstáculos en los puntos ciegos del vehículo o distancias pequeñas, menores a 10 metros en las partes frontal y posterior del vehículo.

Se suelen usar debido a su bajo coste y pequeño tamaño en comparación con los sensores LIDAR.

2.2.5 GPS

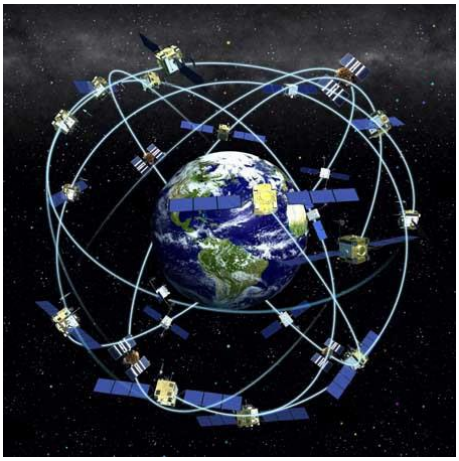

Figura 13: GPS

El sistema GPS o Sistema de Posicionamiento Global, consiste en una red de satélites, desarrollada e instalada por el Departamento de Defensa de Estados Unidos, que permite a cualquier dispositivo con un chip GPS geolocalizarse de manera relativamente precisa.

Existen otros sistemas de posicionamiento menos extendidos como el sistema GLONASS de la federación rusa, o Galileo desarrollado por la Unión Europea. Estos sistemas están en auge, concretamente el sistema GLONASS que promete una mayor precisión, por otra parte, el sistema Galileo está centrando su desarrollo en mejorar la precisión en zonas polares.

En la actualidad cualquier dispositivo electrónico cuenta con un chip GPS. La tecnología actual permite el desarrollo de sistemas GPS de precisión submétrica a un coste asequible. Es un sistema básico en cualquier vehículo autónomo, permite una localización relativamente precisa a pesar de su latencia. Un vehículo puede funcionar de manera relativamente autónoma con solo un GPS teniendo previamente mapeada la carretera y en un entorno controlado, sin obstáculos.

A parte de su uso común suele ser usado como sistema común sincronizador entre los distintos sensores, ya que en su funcionamiento el sistema GPS sincroniza su reloj con el de los satélites los cuales están sincronizados con el reloj atómico. Esto permite a todo el sistema poder estar sincronizado de manera aislada, en cuanto a conexión a internet.

2.2.6 IMU

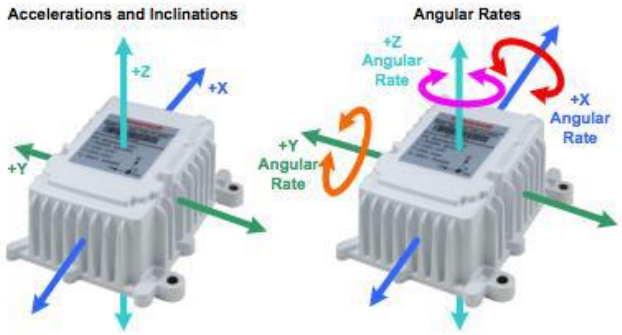

Figura 14: Unidad de movimiento inercial

La Unidad de Movimiento Inercial, IMU, es un dispositivo compuesto por varios acelerómetros y giroscopios mediante los cuales es capaz de medir su velocidad y orientación. Este dispositivo usa los datos de aceleraciones obtenidas por sus sensores internos para realizar un tracking de su movimiento.

Los sensores de uso comercial son precisos en cortos periodos de tiempo, pero no se deben usar durante periodos largos, debido a que tienden a acumular error. Los sensores de uso militar si pueden ser usados durante largos periodos de tiempo, pero su coste es demasiado elevado, por ello se suele optar por usar otros métodos de posicionamiento como principales y reservar este tipo de sensores como método de reserva.

La Unidad de Movimiento Inercial es un sensor básico en cualquier vehículo autónomo. A pesar de no ser usada para realizar mediciones directas, se usan para realizar ajustes a las mediciones de otros sensores, el más usual es el GPS. Esto se debe a que la frecuencia de medición del GPS es mucho más lenta, lo que significa que el vehículo solo supone su posición a partir de las últimas mediciones del GPS.

Se suelen usar filtros tales como el Filtro de Kalman, para hacer más preciso el conocimiento de la posición entre las mediciones del GPS, además de que permite detectar posibles errores en la medición.

2.2.7 Ultrasonido

Figura 15: Sensor de Ultrasonidos

Los sensores de ultrasonido son dispositivos capaces de medir la distancia a una superficie de cualquier material. El único requisito es que reboten el sonido. Estos sensores emiten un sonido en forma de pulso y miden el tiempo que tarda en regresar. Son precisos en distancias de hasta 6 metros.

En el ámbito de los vehículos autónomos se suelen emplean en las zonas ciegas del vehículo. Su uso más extendido es como ayuda al aparcamiento, midiendo la distancia al vehículo u objeto anterior y posterior. Pero también tiene otros ámbitos de uso, por ejemplo, como indicador en la zona ciega del retrovisor.

2.3 Procesamiento de la información

Como se ha visto en el apartado anterior, en un vehículo autónomo se pueden encontrar gran cantidad de sensores. Estos arrojan una enorme cantidad de información por segundo, pero esta información resulta inútil si no se realiza un procesamiento adecuado la misma.

En este apartado se verán los métodos más comunes para el tratamiento de dicha información, así como algunos de los algoritmos más extendidos. También se verán las principales divisiones en cuanto a procesamiento de información que se dan en un vehículo autónomo.

2.3.1 Fusión de sensores

Uno de los principales campos en los que se está centrando el desarrollo en cuanto a procesamiento de información es la fusión de sensores. La fusión de sensores consiste

en procesar de manera conjunta la información que se obtiene de varios sensores. El trabajo de la fusión de sensores es el siguiente:

- Se encargar de recopilar la información proveniente de los sensores.
- Procesa la información de forma inteligente de manera que detecta posibles errores en la adquisición de datos, analiza información redundante proveniente de distintas fuentes y deshecha la información innecesaria.
- Genera un nuevo set de información más completa y precisa del entorno que rodea al sistema en ese momento.

Este proceso es una parte crítica del sistema e influye directamente en la toma de decisiones que se verá posteriormente.

La mayoría de los algoritmos desarrollados para realizar este proceso están basados en la Inferencia Bayesiana que se verá a continuación, aunque también existen otros métodos basados en inteligencia artificial.

2.3.2 Inferencia Bayesiana

La inferencia bayesiana es un tipo de inferencia estadística que se basa en el teorema de Bayes y consiste en inferir la probabilidad de una hipótesis a partir de la probabilidad de varias observaciones. Este tipo de inferencia se aplica en fusión de sensores cuando de dos o más sensores se extrae información redundante, de la cual se puede inferir otra información más precisa o "creíble" en términos de probabilidad.

Como se ha indicado antes existen varios algoritmos basados en la inferencia bayesiana para abordar el problema de la fusión de sensores, algunos de los más comunes son los siguientes:

- Filtro de información, que consiste en eliminar información redundante y manteniendo la de mayor probabilidad.
- Filtro de partículas, que es utilizado para estimar el estado un sistema que cambia en el tiempo.
- Filtro de Kalman, que utiliza una serie de observaciones en el tiempo que tienen cierta incertidumbre y producen estimaciones de variables desconocidas que tienden a ser más precisas que aquellas basadas en una única observación.

2.3.3 Auto localización (SLAM)

La auto localización o SLAM (Simultaneous location and mapping) por sus siglas en inglés, es una técnica que consiste en la construcción de un mapa del entorno a medida que se mueve el vehículo, a su vez le permite localizarse en el entorno y estimar su trayectoria. El principal problema que se busca resolver con esta técnica es dar la capacidad a un vehículo de ser posicionado en un entorno desconocido y que este sea capaz de construir un mapa del entorno y localizarse dentro del mismo.

Esta técnica está directamente relacionada con la inferencia bayesiana, que se ha visto anteriormente, debido al problema de la incertidumbre generada por la inexactitud de los sensores, además de la falta de precisión en cuanto a la medición del movimiento del vehículo. Por ello se usa la inferencia bayesiana para determinar exactamente, o al menos mejorar, la posición de los obstáculos y la localización del vehículo.

Actualmente, existen dos métodos especialmente extendidos para la implementación de esta técnica, el Filtro Extendido de Kalman y los Mapas de Ocupación de Celdillas. Estas dos implementaciones destacan por sus diferencias.

El Filtro Extendido de Kalman usa la inferencia bayesiana para definir de manera más precisa los objetos y obstáculos del entorno, haciendo más precisa la localización, pero la linealizacion característica del Filtro Extendido de Kalman, que es lo que lo diferencia del Filtro de Kalman, provoca que las estimaciones cambien a lo largo del tiempo y no se correspondan con sus valores reales. Como consecuencia de esto la incertidumbre en las estimaciones no se corresponde con el verdadero error cometido. A demás el coste computacional es exponencial respecto al número de objetos. A pesar de todo esto es la implementación más extendida debido a que describe el entorno de manera precisa y es capaz de cerrar bucles.

Los Mapas de Ocupación de Celdillas se basan en discretizar el espacio es parcelas de tamaño fijo que se identifican como ocupadas o vacías con cierta incertidumbre. Esta implementación requiere que la posición del vehículo sea conocida, por lo que se necesita un método auxiliar que estime la posición del vehículo en cada instante. Su precisión permite que el error acumulado sea pequeño, ya que cuanto más discretizado este el espacio mayor será su precisión. Algunas de sus ventajas son: su implementación es sencilla, no requiere definir completamente los objetos, la generación de trayectorias se simplifica debido a la partición del espacio en ocupado o vacío, se puede ajustar la precisión del algoritmo a la potencia computacional disponible y puede ser extendido al espacio tridimensional de manera sencilla.

2.3.4 Detección y evaluación de obstáculos

La detección y evaluación de obstáculos la parte del procesamiento de la información más desarrollada en el tiempo, se han desarrollado múltiples algoritmos para implementar este paso dependiendo del sensor al que se aplique.

Para los sensores cuya información son imágenes como en el caso de cámaras y cámaras estéreo, por ejemplo, se usan métodos de procesamiento de imagen con el objetivo de diferenciar los obstáculos del fondo, también se usan para realizar la segmentación de carreteras e interpretar las señales de tráfico.

Por otra parte, para aquellos sensores que realizan mediciones de distancias como los ultrasonidos, radares y laser scanner se utilizan otros tipos de técnicas como el reconocimiento de objetos mediante algoritmos geométricos, que identifican obstáculos reconociendo su forma, por densidad de puntos como el algoritmo "Voxel grid", que divide el espacio tridimensional en celdillas, además de otros algoritmos de segmentación de nubes de puntos.

Este paso supone una parte critica en el sistema debido, tanto a que su resultado ha de ser lo más preciso posible para asegurar el reconocimiento de todos los potenciales obstáculos de la carretera, como que su tiempo de ejecución ha de ser tiempo real, varios algoritmos de reconocimiento de obstáculos suelen ejecutarse a la vez e independientes. Y posteriormente se suelen realizar nuevamente, algoritmos de fusión de datos entre los resultados de diferentes algoritmos para mejorar el resultado.

En este paso también se realiza el seguimiento y reconocimiento de trayectorias, "tracking", de los obstáculos con el fin de mejorar los algoritmos de detección e identificar el riesgo que suponen para el vehículo.

2.3.5 Planificación de trayectorias

La planificación de trayectorias o "path planning" son estrategias que se encuentran en los coches semiautónomos o autónomos. Estas estrategias consisten en definir la ruta que ha de seguir el vehículo.

La planificación de trayectorias se suele dividir en dos niveles:

- Planificación de la trayectoria global: Que consiste en definir la ruta que va a seguir el vehículo desde el punto inicial hasta el destino.
- Planificación de la trayectoria parcial: Que consiste en definir detalladamente cual va a ser el movimiento del vehículo.

Para abordar este problema se suele usar un GPS o un sistema de planificación de rutas similar, como sistema para planificar la trayectoria global como en la imagen siguiente, en la que se puede ver la ruta definida por Google Maps para llegar a la Escuela Técnica Superior de Ingeniería de Sistemas Informáticos desde el edificio del rectorado de la Universidad Politécnica de Madrid.

Figura 16: Ejemplo de planificación global de trayectorias

Tras haber definido cuál será la ruta a seguir, se definen unos checkpoints que dividen la ruta en segmentos, de longitud variable dependiendo de la ruta, por ejemplo, en una recta los checkpoints suelen estar más separados entre sí, aunque esto depende de la técnica utilizada. En este punto es cuando entra la planificación de trayectorias parcial, esta se encarga de definir la ruta de manera precisa en cada segmento.

En la Figura 17: Aproximación de trayectoria en una curva mediante curvas de Bézier se puede observar como la planificación de trayectorias parcial ha definido la ruta a seguir en un segmento en el que hay una curva.

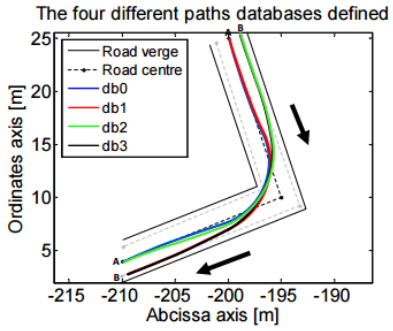

Figura 17: Aproximación de trayectoria en una curva mediante curvas de Bézier

En la imagen posterior se puede observar cómo definiría la ruta con un conjunto de curvas consecutivas, mediante la concatenación de curvas de Bézier.

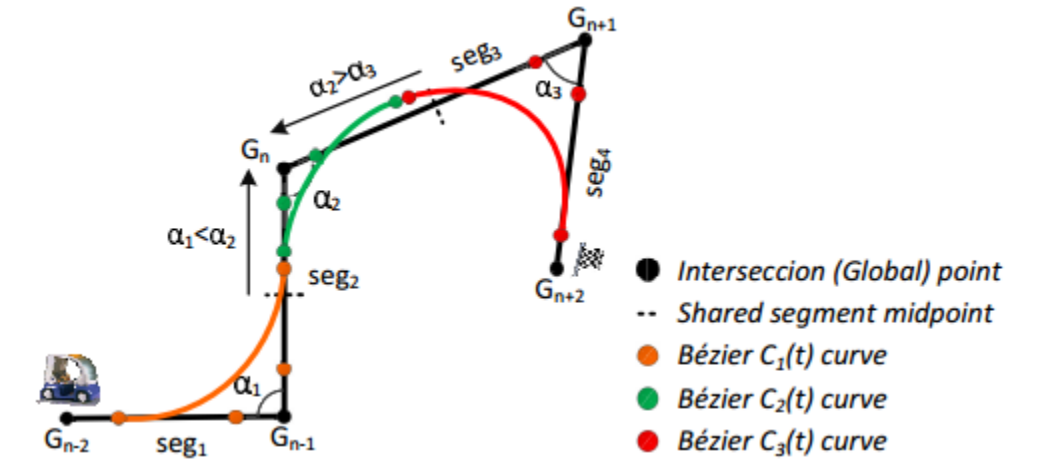

Figura 18: Aproximación de trayectoria en curvas consecutivas mediante curvas de Bézier (6)

Hasta este punto se supone un entorno estático, lo que significa que no hay obstáculos en movimiento. El verdadero problema computacional de estas estrategias es al trabajar en un entorno de tiempo real en el que hay obstáculos móviles.

2.3.6 Evasión de obstáculos

En entornos reales sobre la planificación de trayectorias vista anteriormente se aplican técnicas de evasión de obstáculos, encargadas de modificar la trayectoria original con el objetivo de evitar los obstáculos que el vehículo se pueda encontrar.

Habitualmente se definen dos acciones que puede realizar un vehículo al encontrarse un obstáculo, adelantar o frenar. Existen múltiples técnicas para definir la ruta a seguir en una maniobra de adelantamiento, la mayoría están basadas en el análisis de grafos, en las imágenes siguientes se puede ver un ejemplo del resultado de la implementación de las técnicas de campos potenciales y utilizando lógica difusa.

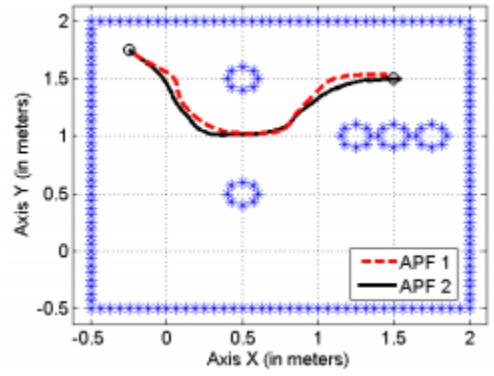

Figura 19: Evasión de obstáculos mediante campos potenciales (7)

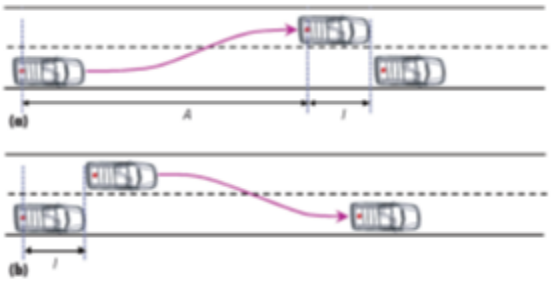

Figura 20: Aproximación de adelantamiento mediante lógica difusa (8)

2.3.7 Toma de decisiones

El último pasó en el procesamiento de la información en un vehículo autónomo es la toma de decisiones. Este paso no se suele ejecutar al mismo nivel que la adquisición y

procesamiento de datos, si no que se da en un nivel superior, por encima de los procesamientos de datos individuales de cada sensor.

En este paso, el proceso encargado de la toma de decisiones evalúa la información obtenida del procesado de la información de cada sensor, así como la información interna del vehículo, para actuar de la mejor manera posible dentro de las capacidades del vehículo. El sistema ha de tener en cuenta las características y limitaciones del vehículo, tales como el peso, el agarre, el ángulo máximo de giro de las ruedas, la capacidad de respuesta del motor, etcétera. A demás de lo anterior, el sistema trabaja siempre bajo unos márgenes de seguridad que aseguran la correcta respuesta del vehículo en condiciones no ideales.

Habitualmente, la toma de decisiones se encuentra fuertemente condicionada con la seguridad de las personas, tanto de los pasajeros como de los peatones u otros vehículos de la calzada, por lo que las técnicas utilizadas para abordar este problema buscan en la medida de lo posible evitar cualquier accidente o en caso de que sea inevitable reducir al máximo los daños. Los sistemas más avanzados, en caso de accidente inevitable, identifican el tipo de obstáculos que encuentra y tratan de evitar los daños humanos por encima de todo. Recientemente, ha surgido una tendencia en cuanto a implementación de estas técnicas, iniciadas por la marca Mercedes Benz en sus vehículos, que anteponen la seguridad de los pasajeros del vehículo aun por encima de la seguridad de los peatones.

2.4 Conducción autónoma

En la actualidad, no se puede decir que exista un vehículo completamente autónomo. Por ello la Sociedad de Ingenieros de Automoción (SAE, Society of Automotive Engineers), la organización encargada del desarrollo de los estándares para vehículos ha desarrollado una clasificación de vehículos autónomos dependiendo de la intervención humana necesaria para su circulación.

2.4.1 Niveles de clasificación de coches autónomos

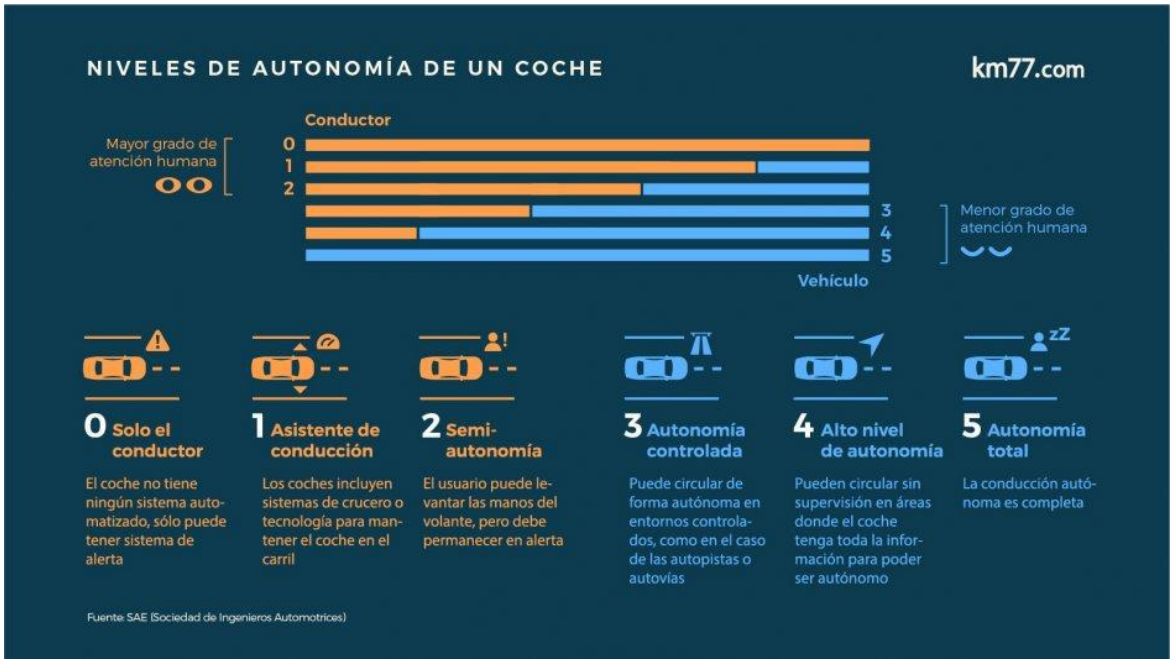

Figura 21: Niveles de autonomía de un vehículo autónomo

Los niveles de clasificación de la autonomía de un vehículo son 6 como vemos en la imagen anterior. A continuación, se va a explicar en qué consiste cada nivel de autonomía de menos autónomo, nivel 0, a completamente autónomo, nivel 5.

Nivel 0

En este nivel se encuentran los vehículos que no cuentan con ningún sistema autónomo ni de ayuda a la conducción o similar, por lo que toda la responsabilidad recae sobre el conductor. Se incluyen en este nivel los vehículos con sistemas de aviso, como los sistemas de asistencia de aparcamiento o avisos de obstáculos en el ángulo ciego del retrovisor.

Nivel 1

En el nivel 1 se encuentran los vehículos con unas capacidades básicas para la conducción, y son capaces de conducir el vehículo por sí mismos. Es el caso de los vehículos que cuentan con sistemas como el control de crucero o que son capaces de mantener el coche en el carril en el que circula. Los vehículos situados en este nivel son capaces de manejar una función primaria, volante o acelerador, por ejemplo, nunca más de una. En este nivel el vehículo continúa necesitando un alto grado de atención por parte del conductor.

Nivel 2

Los vehículos que se pueden encontrar en el nivel 2 reciben la denominación de semiautónomos. En este nivel el vehículo ha de ser capaz de tomar el control de la conducción, a pesar de ello el conductor ha de mantenerse atento para actuar si es

necesario. Los vehículos de este nivel son capaces de manejar varias funciones primarias a la vez.

Nivel 3

En el nivel 3 se encuentran los vehículos capaces de circular en entornos controlados. A este nivel todas las funciones primarias están automatizadas permitiendo al vehículo tener libertad en la conducción. En este nivel el vehículo es capaz de tomar decisiones y realizar maniobras como adelantamientos, aunque el conductor ha de permanecer atento.

Nivel 4

En el nivel 4 ya no se necesita que el conductor esté atento a la conducción. En este punto el vehículo ha de ser capaz de circular de manera totalmente autónoma y resolver cualquier situación en un entorno controlado, como entornos totalmente mapeados.

Nivel 5

En el nivel 5 se encontrarían los vehículos capaces de circular de manera totalmente autónoma en entornos desconocidos. En la actualidad no existe ningún coche que pueda clasificarse en este nivel.

2.4.2 Sistemas autónomos actuales

A continuación, se van a exponer los vehículos autónomos que se pueden encontrar en la actualidad y sus capacidades.

Vehículos semiautónomos actuales

Mercedes Benz

Figura 22: Coche autónomo Mercedes Benz

Desde abril de 2016 está en circulación el clase E de Mercedes. Este vehículo clasificado como semiautónomo de nivel 2. El sistema, que ha de ser accionado por el conductor, mantiene al vehículo en el carril y cuenta con control de crucero. Pero no es capaz de tomar decisiones por su cuenta, por ello se encuentra en el nivel 2.

El vehículo es capaz de cambiar de carril con seguridad si el conductor se lo indica accionando el intermitente, siempre teniendo en cuenta las normas de circulación, como

por ejemplo si hay línea continua no la cruzara, aunque el conductor le indique el cambio de carril. A demás antes de realizar la maniobra realiza una comprobación de seguridad.

El vehículo cuenta con frenada de emergencia automática a velocidades inferiores a 60 km/h. Y velocidades superiores en caso de que un vehículo se cruce de forma transversal. Otra de las innovaciones en este vehículo es que es capaz de detectar si un vehículo se acerca por detrás a demasiada velocidad, en este caso activa las luces de emergencia para indicárselo al vehículo posterior y prepara los sistemas de seguridad tanto activos como pasivos para el impacto, precarga los cinturones, prepara el freno, etcétera.

Waymo, Google

Figura 23: Vehículo autónomo Waymo

Waymo es una empresa que se ha creado especialmente para la rama de vehículos autónomos de la empresa Google, debido a su madurez. Los vehículos de Waymo incorporan la tecnología driverless de Google con la que han obtenido la primera licencia de circulación para un coche autónomo en el estado de Nevada.

Estos vehículos, unos de los más avanzados hasta el momento cuentan con la tecnología necesaria y un software suficientemente avanzado para conducir de manera autónoma. Los vehículos de Waymo se clasifican en el nivel 3 en la escala de autonomía de la SAE.

En la imagen siguiente se puede ver cómo percibe el entorno el vehículo. Mediante sus laser-scanner LIDAR, sus cámaras y cámaras estéreo.

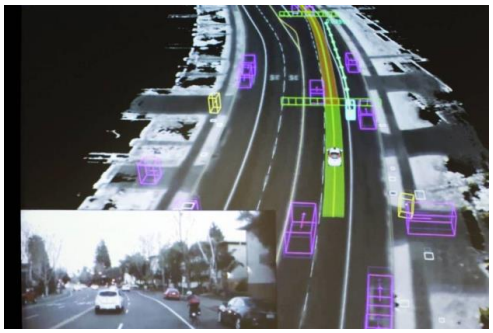

Figura 24Visión del entorno de los vehículos Waymo

Uber

Figura 25: Coche autónomo Uber

Uber ha sido la primera empresa en probar sus vehículos en entornos reales, concretamente en Pittsburg. A pesar de que legalmente los vehículos autónomos de Uber se consideran de nivel 3 debido a que están obligados a tener siempre un conductor, tecnológicamente estarían mejor clasificados en el nivel 4 pese a aun tener fallos, como por ejemplo que no reconocen el carril bici.

En este momento, Uber es la empresa más avanzada en materia de vehículos autónomos a pesar de ser una de las últimas en meterse en este campo. Esto se debe principalmente a su gran flota de vehículos de los que puede extraer gran cantidad de información, ya que este es su entorno de trabajo, al contrario que otras compañías que tienen que hacer grandes campañas de recolección de datos.

La previsión para los próximos años de Uber es, mediante la alianza formada a finales de 2016 con la empresa Volvo, desarrollar un paquete de hardware y software que dote al vehículo de una capacidad totalmente autónoma para circular. A continuación, se puede ver una imagen del prototipo de vehículo autónomo de Uber en conjunto con Volvo.

Figura 26: Prototipo de vehículo autónomo Uber - Volvo

Tesla

Figura 27: Coche autónomo tesla

Algunos de los modelos de vehículos de la marca Tesla incorporan un sistema denominado "autopilot", que legalmente se considera un sistema de ayuda a la conducción, lo que significa que pertenece al nivel 2, pero realmente puede funcionar como un sistema de nivel 3 ya que es capaz de circular de manera relativamente autónoma. El sistema "autopilot" es capaz de, en entornos considerados sencillos como autopistas, mantener al vehículo en su carril, funciona como control de crucero adaptativo que mantiene la velocidad del vehículo en concordancia con la velocidad del resto de vehículos en la calzada. Además, lo que demuestra que es un sistema de nivel 3, es capaz de realizar adelantamientos automáticamente de manera segura.

Las estadísticas que han sido presentadas por la marca demuestran la seguridad del sistema a pesar de estar aun en desarrollo. Las estadísticas de accidentes por kilómetro han sido reducidas un 40% desde la incorporación del sistema en sus vehículos.

En cuanto a tecnología, los vehículos cuentan con 8 cámaras con visión de 360 grados y 12 sensores de ultrasonidos además de un radar en la parte frontal. En la imagen siguiente se puede observar cómo percibe el entorno un vehículo Tesla, siendo el área

amarilla percibida mediante las cámaras y los sensores de ultrasonido, y el área verde por el radar frontal.

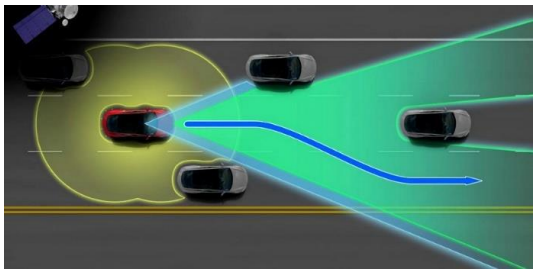

Figura 28: Percepción del entorno de un vehículo Tesla

Por último, una innovación que solo incorpora la marca Tesla es que el sistema está preparado para instalar actualizaciones automáticamente, mediante su sistema "Over-The-Air" que dota de enlace a internet al vehículo permitiendo la instalación de actualizaciones y por otra parte la recolección de datos para la empresa. Mediante este sistema la marca espera convertir, progresivamente mediante las actualizaciones, los vehículos actuales en sistemas de nivel 5.

Baidu

Figura 29: Coche autónomo Baidu

La empresa Baidu al contrario que las otras empresas que están desarrollando vehículos autónomos ha decidido no centrarse en el desarrollo de un vehículo autónomo completo si no en el desarrollo de la tecnología para modificar un vehículo convencional en autónomo.

La primera aproximación de Baidu a la construcción de un vehículo autónomo fue la modificación de un BMW Serie 3 como el que se puede ver en la imagen anterior, con el

que consiguió completar un recorrido de 30 kilómetros de manera completamente autónoma y sin conductor en Pekín en 2015. Donde llego a circular a una velocidad de hasta 100 kilómetros por hora, una velocidad bastante superior que las de otras compañías las cuales no suelen permitir la circulación de un coche completamente autónomo a más de 30 km/h y en limitados casos a 50 km/h.

Los vehículos Baidu funcionan gracias al sistema "Baidu Autobrain" que basa su funcionamiento en mapas del entorno con un alto grado de detalle, por lo que el sistema centra su potencia computacional en el reconocimiento y evaluación de obstáculos. Sus vehículos cuentan con un sensor LIDAR de 64 canales en la parte superior del vehículo, además de varias cámaras de video y radares.

2.4.3 Previsión del desarrollo de los vehículos autónomos

En este apartado se van a exponer las previsiones de algunas marcas en su objetivo de conseguir un vehículo autónomo.

Figura 30: Previsión de desarrollo de los vehículos autónomos

En la actualidad todos los sistemas expuestos anteriormente son considerados como máximo de nivel 2, debido principalmente a las restricciones legales. Se ha expuesto la necesidad de modificar los reglamentos de circulación de los países ya que ninguno contempla la posibilidad de circulación de un coche totalmente autónomo. Los vehículos se suelen usar zonas de pruebas controladas o permisos de circulación especiales para las pruebas de circulación.

En la actualidad, el sistema comercial más avanzando es considerado el sistema "autopilot" de Tesla. Y así se demuestra en su previsión de conseguir alcanzar el desarrollo de un vehículo autónomo de nivel 5 para 2018, lo cual es un objetivo muy ambicioso en comparación a los objetivos del resto de marcas que no esperan un sistema

de nivel 5 antes de 2020 o 2021, incluso en algunas de las marcas más avanzadas en el sector.

En la tabla anterior podemos ver reflejadas algunas marcas de vehículos cuyos sistemas no se han presentado en el apartado anterior. Esto se debe a que estas marcas no esperan un sistema de nivel 5 si no sistemas capaces de intervenir en caso de peligro, los cuales son sistemas de niveles 3 o 4.

En general se espera que, en un plazo de 5 años, alrededor de 2022 los vehículos autónomos sean una realidad. El instituto de ingenieros eléctricos y electrónicos, IEEE por sus siglas en inglés, ha estimado que para el 2040 el 75% de los vehículos serán autónomos.

2.5 Historia de la inteligencia artificial

En este apartado se hará una breve exposición del desarrollo de la inteligencia artificial

2.5.1 Termino de inteligencia artificial

En 1956 aparece por primera vez el término "inteligencia artificial", de la mano de John McCarthy, quien la definió como "La ciencia e ingenio de hacer máquinas inteligentes".

2.5.2 Sistemas expertos

En 1965 la Universidad de Stanford investigó sobre los "sistemas expertos", que son programas especializados que pueden resolver problemas, mediante la toma de decisiones de manera similar a un humano.

Existen varios tipos de sistemas expertos:

- Basados en reglas previamente establecidas
- Basados en casos
- Basados en redes bayesianas

Las principales tareas que realiza un sistema experto son:

- Monitorización
- Diseño
- Planificación
- Control
- Simulación
- Instrucción
- Recuperación de información

2.5.3 Lenguaje Wave

En 1973 fue desarrollado en la universidad de Stanford, el primer lenguaje de programación para robots. Este lenguaje se caracteriza porque permite definir un modelo del entorno del robot para fines de planificación. Este lenguaje es compatible e independiente del robot utilizado.

2.5.4 Agentes inteligentes

Un agente inteligente, es una entidad capaz de percibir su entorno, procesar tales percepciones y responder o actuar en su entorno de manera racional, es decir, de manera correcta y tendiendo a maximizar un resultado esperado. Un agente inteligente puede ser una entidad física o virtual.

Un agente inteligente es un programa especialmente concebido para realizar ciertas tareas de manera autónoma en una red por encargo de un usuario. Esta herramienta permite ganar tiempo en la vigilancia y la colecta de información de interés para la empresa. El usuario define los parámetros de la tarea que realizará de manera autónoma el agente, luego el agente informa de los resultados al usuario.

Tres características fundamentales de los agentes inteligentes:

- La inteligencia: el agente sabe razonar y aprende a partir de la información que recoge
- La interactividad: el agente puede interactuar con su entorno y otros agentes con el propósito de realizar una tarea
- La autonomía: el agente puede tomar decisiones de manera autónoma.

Un agente inteligente se caracteriza por su autonomía en el proceso y el análisis de la información que maneja. Un agente inteligente puede ser controlado por una persona o por otro agente.

2.5.5 Robots

En 2011 la empresa Honda presenta su robot humanoide "Asimo". El nombre de ASIMO es el acrónimo del inglés paso avanzado en movilidad innovadora. Asimo, es un robot humanoide cuyo objetivo de desarrollo es ayudar a las personas que carecen de una movilidad completa.

En la Figura 31: Asimo se puede ver una imagen del robot humanoide Asimo. Este robot destaca por que incluye un control de comportamiento inteligente y autónomo.

Figura 31: Asimo

2.5.6 Inteligencia artificial de Google

Una de las últimas innovaciones en el campo de la inteligencia artificial es la inteligencia artificial de Google. En 2018, Google presento su última evolución en materia de inteligencia artificial, mostrando un sistema inteligente capaz de comunicarse con una persona a través de una llamada telefónica sin que el otro interlocutor se dé cuenta de que no es un humano.

Esto significa que es capaz de:

- Comunicarse en lenguaje natural
- Procesar la información que recoge en forma de voz
- Tomar decisiones y adaptar sus respuestas
- Hacerse pasar por un humano

Esta tecnología se ha denominado Google Duplex. Y es capaz de realizar tareas básicas sin necesidad de que un humano esté implicado. Aunque en caso de que dicha tarea se torne demasiado complicada es capaz de recurrir al usuario para finalizarla.

3 Metodología

En esta sección se van a exponer algunas decisiones de diseño que se han tomado para este proyecto. En este proyecto se pretende desarrollar un sistema de Deep learning basado en redes de convolución que sea capaz de realizar odometría visual sobre los datos recogidos mediante un LIDAR Velodyne de 16 canales y que funcione en un entorno con el framework de Tensorflow, y ejecutándose sobre un hardware basado en GPUs.

3.1 Lenguajes de programación empleados

3.1.1 Python

El lenguaje de programación utilizado ha sido Python. Se ha decidido trabajar en este lenguaje ya que se trata de un lenguaje de prototipado rápido, que además cuenta con una amplia comunidad que proporciona un soporte continuo a cualquier problema que pueda surgir. Otra de las razones por las que se ha elegido este lenguaje es porque es el que se recomienda para la implementación de redes neuronales con Tensorflow.

Se ha utilizado Python 2.7, se han realizado pruebas con Python 3.5 pero causaba demasiados errores de compatibilidad.

3.2 Librerías especializadas

En este apartado se van a introducir las librerías especializadas en Deep Learning que se han utilizado en este proyecto.

3.2.1 Tensorflow

Tensorflow[1] es una librería de código abierto desarrollada para un alto rendimiento en computación numérica. Tiene una arquitectura flexible y permite un desarrollo rápido de computación para plataformas como CPUs, GPUs o TPUs[2]. Y es fácilmente escalable lo que le permite funcionar desde equipos hasta clústers de servidores.

Tiene su origen en los desarrolladores, investigadores e ingenieros de Google Brain con la organización de inteligencia artificial de Google. Esto aporta un soporte muy fuerte para Machine Learning y Deep Learning, aunque la flexibilidad que aporta esta computación es utilizada en muchos otros campos científicos.

[1] https://www.tensorflow.rg/
[2] Unidad de Procesamiento Tensorial

3.3 Entornos de desarrollo

En este apartado se va a exponer y explicar cuál ha sido el entorno de desarrollo utilizado para la realización de este proyecto.

3.3.1 PyCharm

En este proyecto se ha utilizado el entorno de desarrollo PyCharm para desarrollar en Python 2.7 las aplicaciones de odometría visual mediante redes neuronales convolucionales que se han utilizado. Se ha elegido este entorno de desarrollo ya que ya había sido utilizado anteriormente, y ya que incluye todas las facilidades y características necesarias para este proyecto.

En concreto, se ha utilizado la versión de PyCharm Professional ya que está disponible gratuitamente para estudiantes. En la Figura 32: Interfaz PyCharm se puede ver cómo es la interfaz de desarrollo de PyCharm

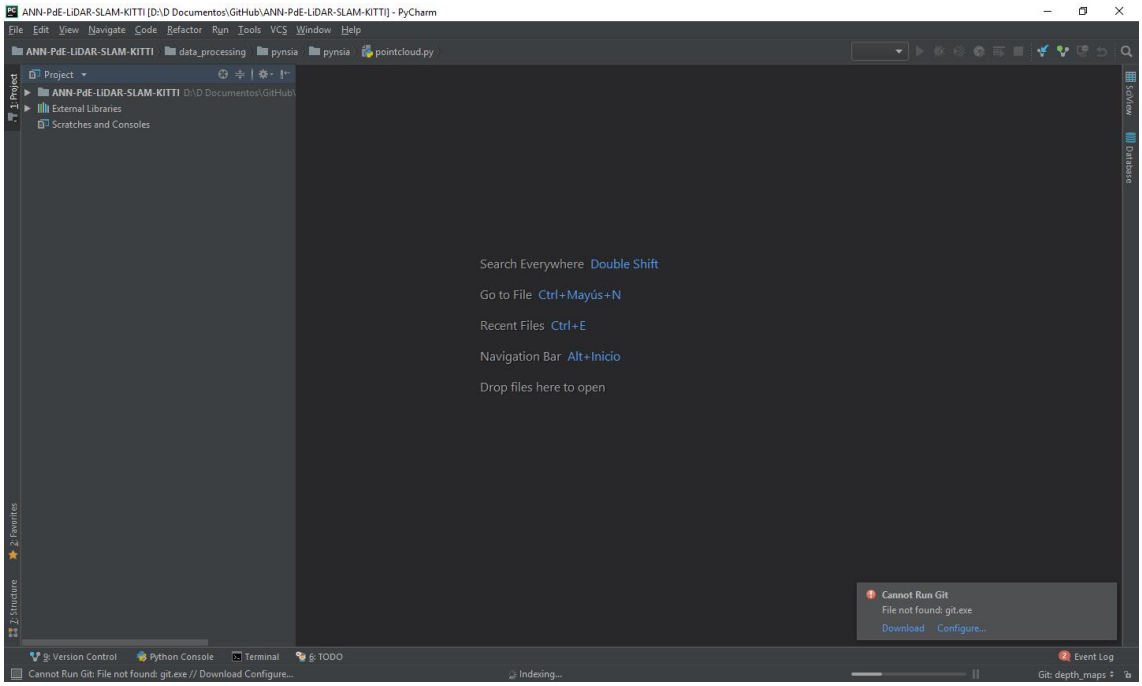

Figura 32: Interfaz PyCharm

3.4 Git

Puesto que el sistema desarrollado era amplio y que más de una persona iba a trabajar en él, se optó por utilizar un sistema de control de versiones. El sistema de control de versiones utilizado fue git. En primer lugar, porque era el sistema conocido y sencillo de manejar, y en segundo lugar porque es el sistema más extendido en el mundo laboral con lo que la experiencia obtenida de su uso sería útil en un futuro.

En un principio, se utilizó el sistema de control de versiones git sobre GitLab en un servidor privado, pero ya que se encontraron problemas en las actualizaciones y que el

INSIA adquirió una licencia de educación para el equipo de investigación se optó por cambiar los repositorios a GitHub.

El uso del control de versiones ha resultado muy útil debido a que ha sido utilizado no solo para el control de versiones si no para llevar un control del avance del proyecto además de ayudar en la detección de errores en el código.

3.5 INSIA

Este proyecto se ha llevado a cabo en el INSIA, Instituto Universitario de Investigación del automóvil.

El INSIA, es un instituto universitario de investigación que pertenece a la Universidad Politécnica de Madrid(UPM) y está directamente relacionado con la Escuela Técnica Superior de Ingenieros Industriales(ETSII).

En el ámbito de los vehículos autónomos este centro es destacado debido a haber sido el primer centro de investigación en este ámbito en España. Este centro cuenta con varios contratos y colaboraciones, tanto civiles como militares, en el ámbito de automoción, vehículos autónomos, comunicación inter-vehicular, etc.

Cabe destacar que el Instituto está acreditado por el Ministerio de Industria español como Servicio Técnico para la homologación en diversos Reglamentos de Naciones Unidas y Directivas Europeas relativas a vehículos, sistemas y componentes de estos, especialmente en el ámbito de la seguridad de autobuses y autocares.

Los principales campos de trabajo desarrollados en el INSIA son:

- Investigación en el área de los vehículos autónomos
- Certificaciones
- Docencia de posgrado y másters de título propio.

El la Figura 33: Panorámica de las instalaciones del INSIA se puede ver una panorámica de las instalaciones del INSIA utilizadas para todas las áreas de investigación.

Figura 33: Panorámica de las instalaciones del INSIA

4 Trabajo Desarrollado

En este apartado se van a desarrollar los tres módulos del trabajo realizado en los que se explicara, en primer lugar, el proceso de instalación de las herramientas y librerías utilizadas. Tras ello se expondrán los módulos de experimentos, resultados y rendimientos tanto software como hardware.

4.1 Instalación

En este primer módulo de instalación se va a explicar la manera de instalar las librerías CUDA y cuDNN, necesarias para la generación de redes neuronales y redes neuronales convolucionales. También se expondrá como crear una maquina en Azure con las características necesarias para entrenar redes neuronales.

Toda la instalación se ha realizado sobre el sistema operativo Ubuntu 16.04 en un ordenador local y Ubuntu Server 16.04 sobre una maquina virtualizada en Azure.

4.1.1 CUDA

CUDA[3] es una arquitectura de cálculo paralelo desarrollada por NVIDIA cuyo objetivo es aprovechar la ponencia de procesamiento paralelo mucha mayor en la GPU que en la CPU. CUDA (9) es utilizado en muchos ámbitos en la actualidad, es utilizado en el ámbito académico para para acelerar AMBER que se utiliza para descubrir nuevos medicamentos, o en el mercado financiero con Numerix y CompatibL que se utilizan para el cálculo de riesgos en bolsa.

En el ámbito que afecta a este trabajo, los vehículos autónomos, CUDA se utiliza para muchas tareas como pueden ser odometría visual, reconocimiento de objetos o segmentación de suelo.

4.1.2 cuDNN

cuDNN[4] es una librería para aceleración de GPU utilizada para redes neuronales profundas. cuDNN (10) facilita las implementaciones de rutinas para convoluciones con propagación hacia adelante y atrás. cuDNN forma parte del "NVIDIA Deep Learning SDK".

cuDNN permite a los desarrolladores centrarse en la implementación de las aplicaciones software en ved de tener que ajustar el rendimiento de la GPU a bajo nivel

[3] http://www.nvidia.es/object/cuda-parallel-computing-es.html
[4] https://developer.nvidia.com/cudnn

que resulta mucho más tedioso y complicado. A demás, cuDNN se puede acelerar aún más utilizando frameworks específicos para Deep Learning como Caffe2 o Tensorflow.

4.1.3 Preinstalación

Limpieza de antiguas instalaciones

Para realizar una correcta instalación se ha de realizar una limpieza de todos los drivers relacionados con NVIDIA que contiene la máquina.

Para ello se utilizan los siguientes comandos:

```
$ sudo apt purge nvidia-*
$ sudo apt purge CUDA*
$ sudo apt autoremove CUDA
```

Se comprueba que no hay repositorios de NVIDIA en /etc/apt/sources.list mediante el comando:

```
$ cat /etc/apt/sources.list
```

4.1.4 Instalación CUDA

Para poder utilizar tensorflow en su opción de GPU se requiere CUDA (11), concretamente CUDA 9.0 ya que CUDA 9.1 no es compatible.

Preinstalación

Antes de proceder a instalar CUDA se ha de verificar que el sistema donde se va a instalar lo soporta

Verificar que se cuenta con una GPU que soporta CUDA

Para ello se utiliza el comando:

```
$ lspci | grep -i nvidia
```

El cual debe mostrar una serie de líneas, lo que significa que si se tiene un GPU utilizable. En caso de que no aparezca nada significara que CUDA no es instalable en este dispositivo.

Verificar que se cuenta con una versión y distribución de Linux que soporte CUDA

Se debe verificar que se está utilizando una versión y distribución de Linux que soporte CUDA, las cuales se encuentran en CUDA Toolkit reléase notes

Para ello se utiliza el comando:

```
$ uname -m && cat /etc/*release
```

Verificar que gcc está instalado

Para comprobar que se tiene instalado gcc se utiliza el comando:

```
$ gcc --version
```

Si se obtiene un mensaje de error es que no está instalado con lo que se tendrá que instalar mediante el comando:

```
sudo apt install gcc
```

Verificar que el sistema cuenta con las cabeceras de kernel correctas y los paquetes de desarrollo instalado

Para el caso de Ubuntu, que es sobre el que se ha trabajado el comando para instalar estas dependencias es:

```
$ sudo apt install linux-headers-$(uname -r)
```

Instalación

Para realizar la instalación de CUDA se debe descargar el CUDA Toolkit 9.0 [5]de la página de descarga oficial de nvidia[6]. Seleccionando las opciones necesarias en nuestro caso. Como se puede ver en la Figura 34: Selección de plataforma para la descarga de CUDA en la que se selecciona una plataforma con Ubuntu 16.04 con el método de instalador mediante un paquete Debian local.

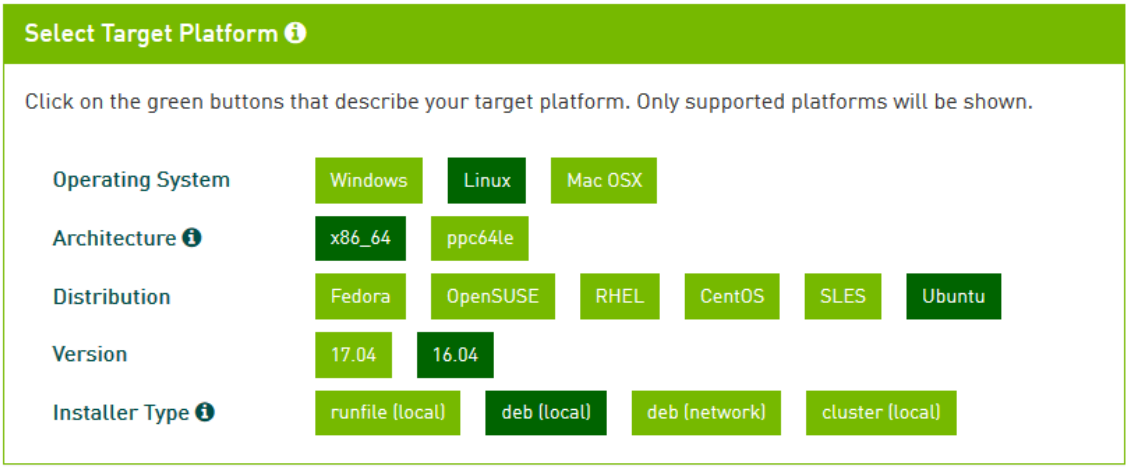

Figura 34: Selección de plataforma para la descarga de CUDA

La descarga se realiza mediante los siguientes dos comandos:

[5] https://developer.nvidia.com/compute/cuda/9.0/Prod/local_installers/cuda-repo-ubuntu1604-9-0-local_9.0.176-1_amd64-deb

[6] https://developer.nvidia.com/cuda-90-download-archive?target_os=Linux&target_arch=x86_64&target_distro=Ubuntu&target_version=1604&target_type=deblocal

```
$ wget https://developer.nvidia.com/compute/cuda/9.0/Prod/local_installers/cuda-repo-ubuntu1604-9-0-local_9.0.176-1_amd64-deb

$ wget https://developer.nvidia.com/compute/cuda/9.0/Prod/patches/1/cuda-repo-ubuntu1604-9-0-local-cublas-performance-update_1.0-1_amd64-deb
```

Una vez descargada, mediante los siguientes comandos se realiza la instalación.

```
$ sudo dpkg -i cuda-repo-ubuntu1604-9-0-local_9.0.176-1_amd64.deb

$ sudo apt-key add /var/cuda-repo-9-0-local/7fa2af80.pub

$ sudo apt-get update

$ sudo apt-get install cuda
```

También se han de instalar los parches[7] que hayan salido hasta el momento actual, en este caso solo hay un parche que se puede descargar de la misma página de descarga del Toolkit.

Estos parches se instalan de la misma manera que en el caso anterior, en nuestro caso mediante el comando:

```
$ sudo dpkg -i cuda-repo-ubuntu1604-9-0-local-cublas-performance-update_1.0-1_amd64.deb
```

Observaciones

Se ha de instalar la versión 9.0, esto se debe a que la versión 9.1, más reciente disponible actualmente, presenta problemas de compatibilidad con el entorno grafico lo que causa errores en el arranque que provocan que no se pueda realizar el entrenamiento.

Se han de realizar los siguientes dos comandos después de la descarga del Toolkit y el parche:

```
$ mv cuda-repo-ubuntu1604-9-0-local-cublas-performance-update_1.0-1_amd64-deb cuda-repo-ubuntu1604-9-0-local-cublas-performance-update_1.0-1_amd64.deb

$ mv cuda-repo-ubuntu1604-9-0-local_9.0.176-1_amd64-deb cuda-repo-ubuntu1604-9-0-local_9.0.176-1_amd64.deb
```

Esto se debe a que mediante el método de descarga que se ha utilizado no se puede ejecutar el comando dpkg directamente debido a que los ficheros no son archivos .deb.

[7] https://developer.nvidia.com/compute/cuda/9.0/Prod/patches/1/cuda-repo-ubuntu1604-9-0-local-cublas-performance-update_1.0-1_amd64-deb

Tras esta instalación se debe realizar un reinicio para que los cambios se hagan efectivos, mediante el comando:

```
$ sudo reboot
```

Debido a que CUDA depende directamente de los drivers gráficos de NVIDIA estos se instalaran automáticamente, por lo que a partir de ahora se dispondrá de entorno gráfico. Pero esta instalación genera un error que consiste en que no se puede iniciar sesión en la interfaz gráfica, pero si en las consolas de Crtl+alt+F1 a Crtl+alt+F6. Para solucionar este error se ha de realizar el siguiente comando:

```
$ sudo apt install –reinstall ubuntu-desktop
$ sudo reboot
```

Verificación

Para finalizar la instalación de CUDA, se realiza una verificación de la correcta instalación.

Verificar la versión del driver

Una vez instalado CUDA se debe comprobar si el driver está cargado mediante el comando

```
$ cat /proc/driver/nvidia/version
```

Verificación del funcionamiento

También se puede comprobar si el path está bien apuntado hacia CUDA, lo cual se puede comprobar mediante el comando

```
$ nvcc -V
```

Lo que devolverá la versión de CUDA que se está utilizando.

4.1.5 Instalación de cuDNN

Una vez se ha instalado CUDA que provee de las operaciones necesarias para la implementación de redes neuronales, se ha de instalar cuDNN (12) para poder utilizar instrucciones necesarias para la compilación de Deep Learning, necesaria para el desarrollo de redes neuronales convolucionales que se han utilizado en este proyecto para el cálculo de la odometría visual.

Descarga de cuDNN

El primer paso para la instalación de cuDNN es descargar los paquetes que la componen. Los paquetes que se han de descargar son los de cuDNN 7.0.5 (versión 7.0.X), ya que se ha comprobado que cuDNN 7.1.X presenta problemas de compatibilidad, concretamente los desarrollados para CUDA 9.0 que es la que se ha instalado en el apartado anterior.

Para descargarlos utilizaremos los siguientes comandos

```
$ wget https://developer.nvidia.com/compute/machine-learning/cudnn/secure/v7.0.5/prod/9.0_20171129/Ubuntu16_04-x64/libcudnn7_7.0.5.15-1+cuda9.0_amd64

$ wget https://developer.nvidia.com/compute/machine-learning/cudnn/secure/v7.0.5/prod/9.0_20171129/Ubuntu16_04-x64/libcudnn7-dev_7.0.5.15-1+cuda9.0_amd64

$ wget https://developer.nvidia.com/compute/machine-learning/cudnn/secure/v7.0.5/prod/9.0_20171129/Ubuntu16_04-x64/libcudnn7-doc_7.0.5.15-1+cuda9.0_amd64
```

Una vez descargados se modificarán para que sean interpretados como paquetes deb mediante los siguientes comandos

```
$ mv libcudnn7_7.0.5.15-1+cuda9.0_amd64 libcudnn7_7.0.5.15-1+cuda9.0_amd64.deb

$ mv libcudnn7-dev_7.0.5.15-1+cuda9.0_amd64 libcudnn7-dev_7.0.5.15-1+cuda9.0_amd64.deb

$ mv libcudnn7-doc_7.0.5.15-1+cuda9.0_amd64 libcudnn7-doc_7.0.5.15-1+cuda9.0_amd64.deb
```

Instalación

Una vez se tienen los paquetes preparados se instalan mediante los siguientes comandos

```
$ sudo dpkg -i libcudnn7_7.0.5.15-1+cuda9.0_amd64.deb

$ sudo dpkg -i libcudnn7-dev_7.0.5.15-1+cuda9.0_amd64.deb

$ sudo dpkg -i libcudnn7-doc_7.0.5.15-1+cuda9.0_amd64.deb
```

Mediante los anteriores comandos se ha instalado la librería cuDNN, la cuDNN para desarrollo y los archivos de pruebas para la comprobar la correcta instalación.

Verificación

Para verificar la correcta instalación se deben realizar los siguientes comandos, mediante los cuales se utilizan unos ficheros de pruebas proporcionados por NVIDIA.

```
$ cp -r /usr/src/cudnn_samples_v7/ $HOME

$ cd $HOME/cudnn_samples_v7/mnistCUDNN

$ make clean && make

$ ./mnistCUDNN
```

Si el resultado de los comandos anteriores es:

```
Test passed!
```

Significa que cuDNN ha sido instalado correctamente.

Observaciones

Es posible que durante la ejecución de los archivos de prueba no arroje un error, pero durante el entrenamiento de una red neuronal sí que ocurra, este error, en caso de que todo esté instalado con corrección, posiblemente habrá sido generado por la utilización de una versión de cuDNN que se está ejecutando en modo de compatibilidad a otra versión. Esto provocara que el Core de TensorFlow se cierre, por lo que es necesario instalar la versión a la que está realizando la compatibilidad.

4.1.6 Montar maquina Azure

En este apartado se va a mostrar cómo se ha realizado la creación, montaje e instalación de una máquina virtual Azure con acceso a varias tarjetas gráficas.

Pasos para crear la máquina virtual

1. Estando en el portal de administración de Azure[8] se selecciona la opción "Crear un recurso"
2. A continuación, dentro del submenú "Proceso" se selecciona la opción "Ubuntu Server 17.10".
3. En este momento se deben indicar los parámetros básicos de la máquina, como son:

[8] https://portal.azure.com

1. El nombre que se va a utilizar en la maquina
2. El tipo de disco que se quiere utilizar. En este punto se debe seleccionar disco HDD ya que las maquinas con discos SSD no cuentan con tarjetas gráficas asociadas.
3. El nombre de usuario que se va a utilizar en la máquina, con su correspondiente contraseña o clave publica para la conexión mediante ssh.
4. El método de pago que se va a utilizar.
5. Se ha de crear o asignar un grupo de recursos[9], un grupo de recursos es una colección de recursos que comparten el mismo ciclo de vida.
6. Por último, se debe seleccionar una ubicación en la que alojar la máquina virtual, se ha de tener en cuenta que no todas las ubicaciones disponibles contienen el mismo tipo de máquinas, en este trabajo se ha utilizado la ubicación "Oeste de Europa".

El resultado de completar estos campos debe ser similar a la Figura 35: Creación VM Azure(Básico).

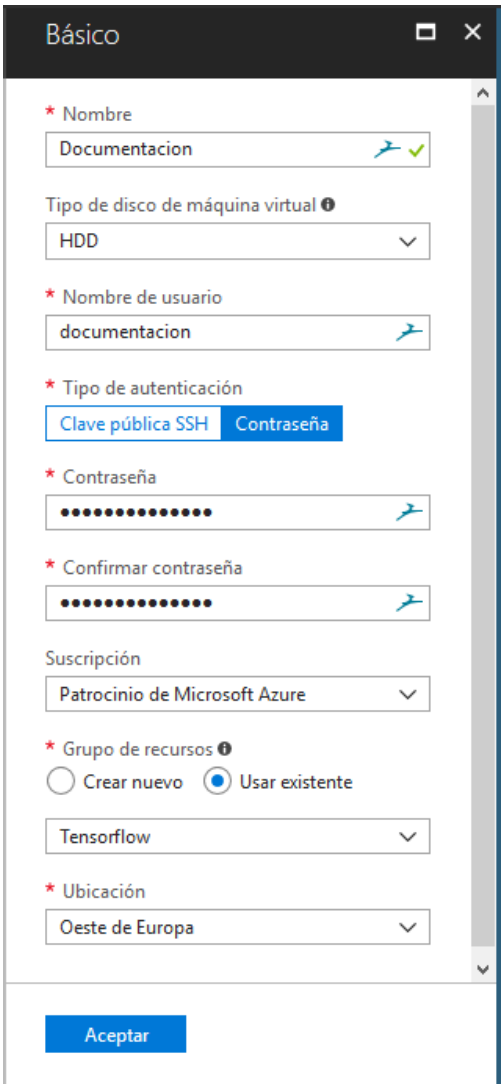

Figura 35: Creación VM Azure(Básico)

4. A continuación, se debe indicar el Tamaño de la máquina que se va a utilizar, en esta sección se elige una de las configuraciones de sistema que Azure tiene preprogramadas, estas configuraciones incluyen el número de procesadores, el tamaño de la memoria RAM, el número de discos de datos y su tamaño, y el equilibrio de carga y las zonas de carga habilitadas.
En el ámbito de este trabajo se requieren maquinas que cuenten con tarjetas gráficas para la ejecución de redes neuronales convolucionales. Las máquinas que cuentan con tarjetas gráficas son las de la serie N[10].
Las maquinas disponibles son las que se pueden encontrar en la Figura 36: Maquinas disponibles.

[9] https://go.microsoft.com/fwlink/?linkid=394393
[10] https://azure.microsoft.com/es-es/pricing/details/virtual-machines/series/

Para este ejemplo se creará una máquina de tipo NV6.

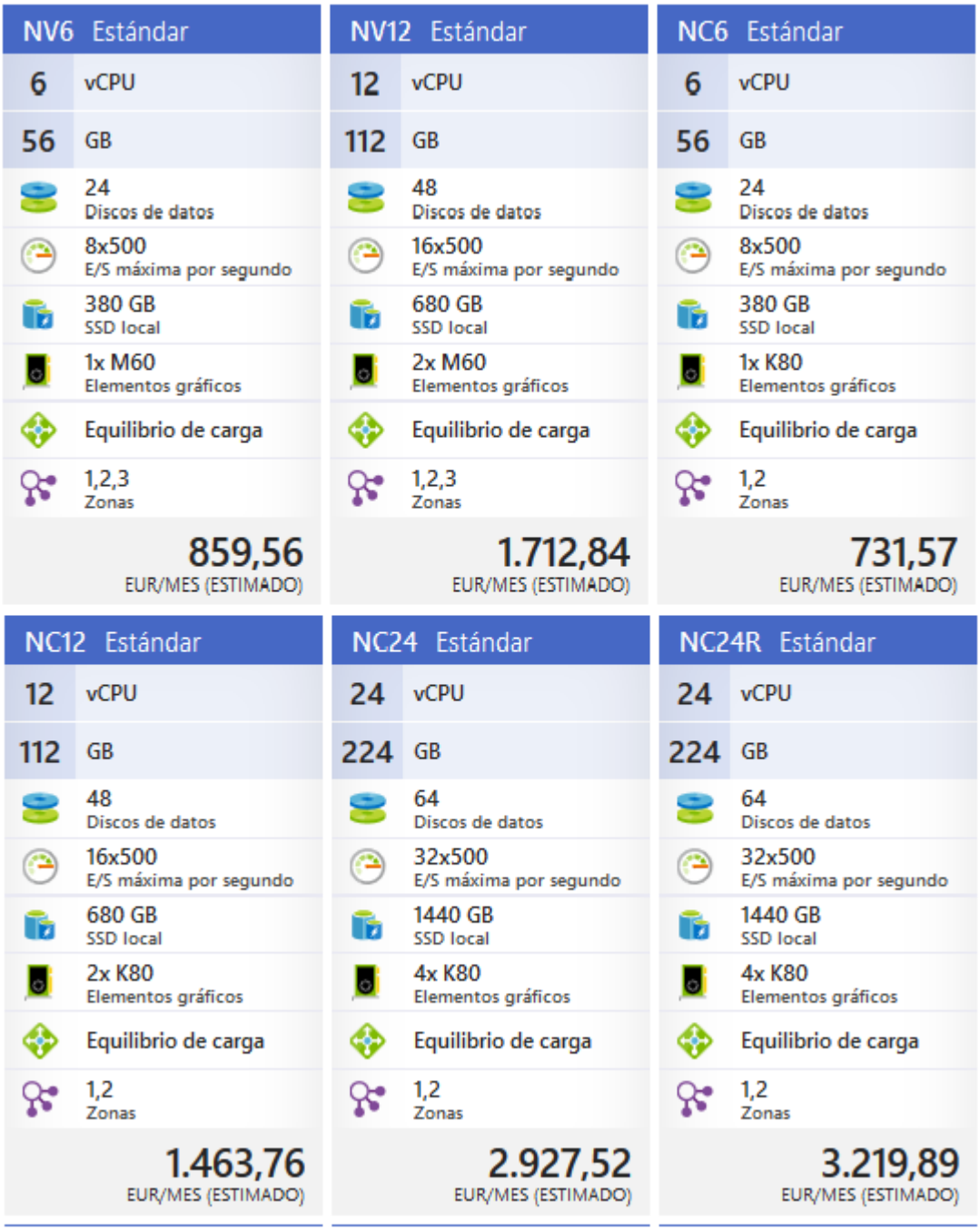

Figura 36: Maquinas disponibles

5. Una vez seleccionado el modelo de máquina que se va a utilizar se ah de configurar los parámetros de red y si se quiere añadir alguna extensión a la máquina, las extensiones son módulos que se instalan automáticamente y le dan funcionalidades especificas a la máquina.
   En este caso se mantendrá la configuración generada por defecto por Azure. Ya que el único parámetro interesante, la red virtual a la que se conecta la maquina ya está definida por defecto al estar en un grupo de trabajo específico. Este parámetro es interesante ya que más adelante en el trabajo se tratará de distribuir la computación de tensorflow entre varias máquinas, para lo cual deben estar conectadas.

6. Por último, Azure mostrara el resumen de la máquina que va a crear, este resumen incluye todos los parámetros que se han configurado en los pasos anteriores, desde el tipo de maquina seleccionada hasta la configuración de la red virtual a la que se conecta.
un punto importante en este apartado es cuando es posible descargar la plantilla de la máquina virtual. Esta plantilla constituye una definición en formato JSON, en la cual están incluidas todas las características configurables, con el fin de facilitar la replicación de la maquina creada.

Observaciones

Por defecto los puertos de conexión de la maquina están cerrados, en este proyecto necesitaremos los puertos de ssh, http y https para la utilización de la herramienta Tensorboard (13). Para abrir dichos puertos se puede hacer de dos maneras.

- Manualmente: Si se quiere abrir dichos puertos una vez creada e instalada la máquina, se ha de entrar en su configuración en el apartado de Redes. Ahí se pueden agregar reglas de entrada al firewall para permitir el tráfico entrante y saliente a dichos puertos.
- Automáticamente: Se puede configurar la maquina durante la instalación para que agregue las reglas del firewall automáticamente para los protocolos seleccionados.

Creación de máquina virtual Azure desde plantilla

Como se puede ver en el apartado anterior, montar una maquina Azure no es difícil, pero es un proceso relativamente largo. Azure cuenta con una funcionalidad que permite crear una máquina virtual rápida y cómodamente a partir de una plantilla.

Una plantilla no es más que una definición en formato JSON de todos los parámetros y características que se han configurado en el apartado anterior. A continuación, se puede ver un fragmento de esta configuración para la máquina que se ha instalado antes.

```
{
      "name": "[parameters('virtualMachineName')]",
      "type": "Microsoft.Compute/virtualMachines",
      "apiVersion": "2017-03-30",
      "location": "[parameters('location')]",
      "dependsOn": [
            "[concat('Microsoft.Network/networkInterfaces/',
parameters('networkInterfaceName'))]"
      ],
      "properties": {
            "osProfile": {
```

```
                "computerName":
"[parameters('virtualMachineName')]",
                "adminUsername": "[parameters('adminUsername')]",
                "adminPassword": "[parameters('adminPassword')]"
            },
            "hardwareProfile": {
                "vmSize": "[parameters('virtualMachineSize')]"
            },
            "storageProfile": {
                "imageReference": {
                    "publisher": "Canonical",
                    "offer": "UbuntuServer",
                    "sku": "16.04-LTS",
                    "version": "latest"
                },
                "osDisk": {
                    "createOption": "fromImage",
                    "managedDisk": {
                        "storageAccountType": "Standard_LRS"
                    }
                },
                "dataDisks": []
            },
            "networkProfile": {
                "networkInterfaces": [
                    {
                        "id":
"[resourceId('Microsoft.Network/networkInterfaces',
parameters('networkInterfaceName'))]"
                    }
                ]
            },
            "diagnosticsProfile": {
```

```
                "bootDiagnostics": {
                    "enabled": true,
                    "storageUri":
"[reference(resourceId('Tensorflow',
'Microsoft.Storage/storageAccounts',
parameters('diagnosticsStorageAccountName')), '2015-06-
15').primaryEndpoints['blob']]"
                }
            }
        }
```

En este fragmento, se pueden ver algunos de los parámetros configurados. Por ejemplo, aparece el sistema operativo que tendrá la máquina, Ubuntu Server 16.04-LTS.

A continuación, se verá cómo implementar esta plantilla una máquina.

En primer lugar, cuando se descarga la plantilla el fichero que se obtiene es un fichero zip con el nombre de la máquina. Este fichero contiene varios archivos, los cuales permiten diferentes modos de implementación de la máquina. Los archivos en los que nos vamos a centrar son "parameters.json" y "template.json". El archivo "parameters.json" únicamente incluye algunos de los parámetros configurados durante la instalación como las redes virtuales a las que se conecta. Este fichero se utiliza para configurar automáticamente los parámetros de una maquina ya creada.

El fichero que realmente se utiliza es el "template.json" el cual incluye tanto los parámetros mencionados anteriormente como los recursos a los que tiene acceso la máquina.

Para crear la maquina a partir de la plantilla se ha de seleccionar la herramienta "implementación de plantillas", dentro del grupo crear un recurso. Una vez aquí se ha de seleccionar "Cree su propia plantilla en el editor". En este momento se tendrá acceso al editor que ya viene con una plantilla básica. En la Figura 37: Editor de plantillas se puede ver cómo es la interfaz del editor.

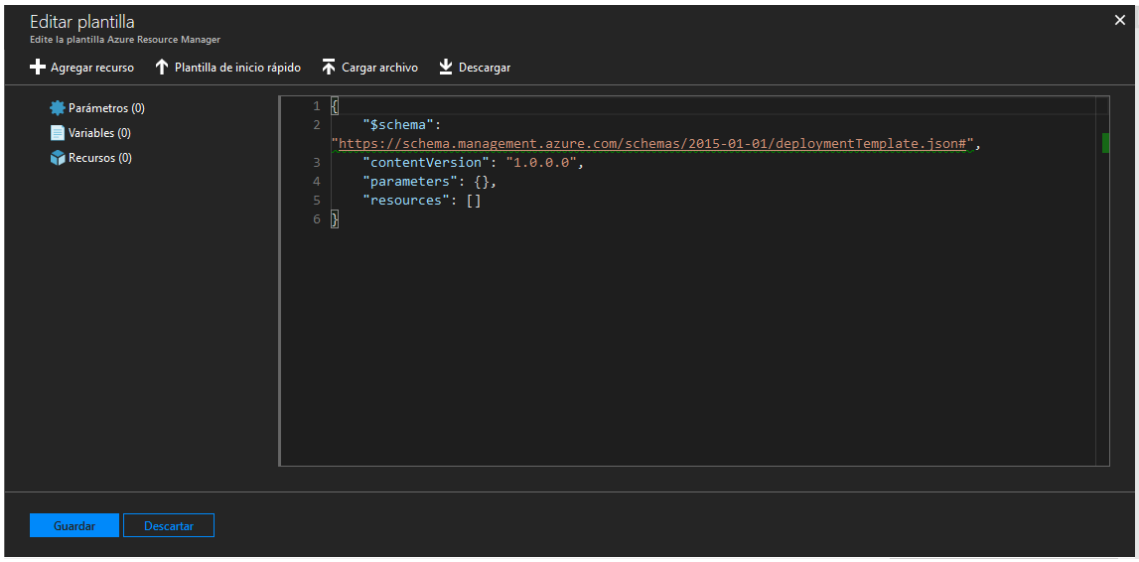

Figura 37: Editor de plantillas

Ahora, se ha de cargar el archivo mencionado anteriormente "template.json". Al cargarlo, se deberán obtener en la interfaz una serie de parámetros, variables y recursos que serán los que definirán la máquina.

Ahora que la plantilla ya está cargada, se ha de guardar en Azure y se podrá implementar la maquina tras incluir algunas configuraciones que se solicitan en el paso siguiente.

4.2 Software

En este apartado se van a exponer cuales han sido las implementaciones software desarrolladas para la realización del proyecto. Este apartado se va a dividir en 2 fases.

Una primera fase, en la que se va a exponer y explicar el código fuente utilizado para las pruebas en una maquina local, el cual se ha utilizado el mismo con diferentes configuraciones de hardware para poder observar los diferentes comportamientos de los recursos.

En la segunda fase se va a exponer una de las versiones más recientes del código fuente que ha sido desarrollado durante el transcurso de este proyecto, el cual trata de implementar una red neuronal convolucional que, mediante datos obtenidos de un sensor LIDAR, sea capaz de realizar una odometría visual, tanto de velocidad como de inclinación de los ejes.

4.2.1 Software para la maquina local

En este apartado se va a explicar detalladamente el código utilizado para esta fase del proyecto. En la cual se ha desarrollado una red neuronal para odometría visual, este fue el primer paso para el desarrollo posterior del proyecto y la introducción de redes neuronales convolucionales.

Los datos utilizados para esta fase del proyecto han sido obtenidos manualmente, mediante un Velodyne LIDAR de 16 capas circulando por los alrededores del Campus Sur de la UPM y a lo largo de un tramo de la autovía A3.

Estos datos han sido previamente pre-procesados, consiguiendo una estructura fija, similar a una imagen de tamaño 300x16. En esta especie de imagen los píxeles en vez de estar conformados por los valores RGB habituales, están formados por valores de distancia al punto, tomando como punto de referencia (0,0,0) la posición del sensor. También se han realizado pruebas dividiendo estos puntos en coordenadas cartesianas (x, y, z).

A continuación, se van a explicar algunas de las partes principales del código, así como la arquitectura de la red implementada.

El código comienza con la inicialización de parámetros, en los que se incluye directorios de logs, variables temporales para trazas, etc.

A continuación, se configura e inicia la sesión de Tensorflow mediante los siguientes comandos.

```
config = tf.ConfigProto(
      # device_count = {'GPU': 0},
      gpu_options=tf.GPUOptions(per_process_gpu_memory_fraction=0.9)
)
sess = tf.Session(config=config)
```

En este fragmento de código cabe destacar el parámetro gpu_options, que le indica a Tensorflow el porcentaje máximo de memoria que se le permite utilizar. En este caso ha sido configurado a un 90 %, ya que se ha comprobado que sin esta opción no existe límite para la memoria de Tensorflow y si se llegase a utilizar el 100% las gráficas se quedan bloqueadas, provocando fallos en otros sistemas.

Ahora que Tensorflow ya ha sido iniciado, se definen los placeholder, que son los puntos de entrada y salida de datos hacia Tensorflow. Estos se definen de la siguiente manera.

```
x = tf.placeholder(tf.float32, shape=[None, ni*nj*ch])
y_ = tf.placeholder(tf.float32, shape=[None, no])
```

Por estos placeholders se introducirán los datos de entrada para el aprendizaje y los datos de validación.

A continuación, se define la arquitectura de la red, que se ha dividido en 6 capas principales.

```
print 'First layer'
W_conv11 = weight_variable([3, 15, ch, f1])
b_conv11 = bias_variable([f1])
h_conv11 = tf.nn.relu(conv2d(x_image, W_conv11) + b_conv11)
h_conv11n = tf.nn.local_response_normalization(h_conv11)
W_conv1 = weight_variable([3, 15, f1, f1])
b_conv1 = bias_variable([f1])
h_conv1 = tf.nn.relu(conv2d(h_conv11n, W_conv1) + b_conv1)
h_conv1n = tf.nn.local_response_normalization(h_conv1)
h_pool1 = max_pool_2x2(h_conv1n) # max pool
```

La primera capa consta de las subcapas que se pueden ver en código anterior. Estas comienzan con la definición de variables de pendiente y desplazamiento. A continuación, se introduce una capa de activación RELU junto con una convolución de 2 dimensiones. Y se realiza una normalización. Por último, se repite el proceso anterior añadiéndole al final una capa de pooling.

La segunda y tercera capa son similares a la anterior. A continuación, se han implementado 3 capas full connected que se han definido de la siguiente manera.

```
print 'First FC layer'
fcin = ninputs / ch / 4 / 4 / 3
W_fc1 = weight_variable([fcin * f3, nfc])
b_fc1 = bias_variable([nfc])
h_pool3_flat = tf.reshape(h_pool3, [-1, fcin * f3])
h_fc1 = tf.nn.relu(tf.matmul(h_pool3_flat, W_fc1) + b_fc1)
keep_prob = tf.placeholder(tf.float32)
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
```

Esta capa full connected, comienza de la misma manera que las primeras capas definiendo las variables de pendiente y desplazamiento. A continuación, se realiza un reajuste del tamaño del tensor de entrada y se introduce una capa de activación RELU sobre el cálculo de función de la recta. Por último, se define un nuevo placeholder para introducir un valor de probabilidad utilizada en la capa de dropout siguiente.

La segunda capa full connected es similar a la anterior pero la tercer y ultima capa full conected únicamente realiza la función de cálculo de la recta para extraer la salida sin normalizar. Esta última capa se define de la siguiente manera.

```
print 'Third FC layer'
W_fc3 = weight_variable([nfc2, no])
b_fc3 = bias_variable([no])
y = tf.matmul(h_fc2_drop, W_fc3) + b_fc3
```

Una vez definida la arquitectura, se define la función de optimización y la función de error. En este caso se utiliza el algoritmo AdamOptimizer como optimizador y para el cálculo del error se han realizado experimentos mediante entropía cruzada y error cuadrático medio, obteniendo resultados similares. También se deben inicializar las variables globales para que estas se carguen en Tensorflow.

```
#cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y_conv),
reduction_indices=[1]))
vRMS = tf.sqrt(tf.reduce_mean(tf.square(y_ - y), reduction_indices=0))
loss = tf.reduce_mean(vRMS)
train_step = tf.train.AdamOptimizer(1e-5).minimize(loss)
sess.run(tf.global_variables_initializer())
```

Por último, ya en la fase de aprendizaje, se ha implementado una función llamada minibatch que se encarga de dividir los datos de entrada en conjuntos manejables y ejecutar el grafo definido anteriormente.

El proceso de aprendizaje es el siguiente:

```
for i in range(1,ite+1):
  minibatch(train_step, net.inputs, net.outputs, batch_size, p)
  if i%trace == 0:
    print '\nEpoch',i
    # calculamos el RMS y lo escribimos por pantalla
    lerrl = minibatch(vRMS, net.inputs, net.outputs, batch_size, 1.0)
    lerr = meanvector(lerrl)
    terrl = minibatch(vRMS,net.inputst, net.outputst, batch_size, 1.0)
    terr = meanvector(terrl)
    print 'Learn RMS:',lerr
```

```
    print ' Test RMS:',terr
    f3.write("Learn RMS = " + str(lerr)+"\n")
    f3.write("Test RMS = " + str(terr)+"\n")
```

En él, se realizan un numero de iteraciones, previamente definidas y se realiza un entrenamiento, que consiste en ejecutar el optimizador del error que se ha comentado con anterioridad.

Y a continuación se calculan los errores de aprendizaje y de test. Que son utilizado interiormente por Tensorflow para reajustar la red neuronal.

También se ha utilizado una función de Tensorflow que resulta especialmente útil, que es el saver. Esta función es la encargada de guardar el estado de la red, y de cargarlo cuando se realiza una ejecución posterior de manera que la red no deba ser entrenada desde 0 en cada ejecución. Y también sirve para generar trazas durante la ejecución de la red que permitan volver a ejecutarla desde un estado anterior, por ejemplo, en caso de que se detecte overfitting.

En la Figura 38: Grafo red neuronal local se puede observar un gráfico conceptual que se ha realizado para una visualización más simple de la arquitectura de la red implementada.

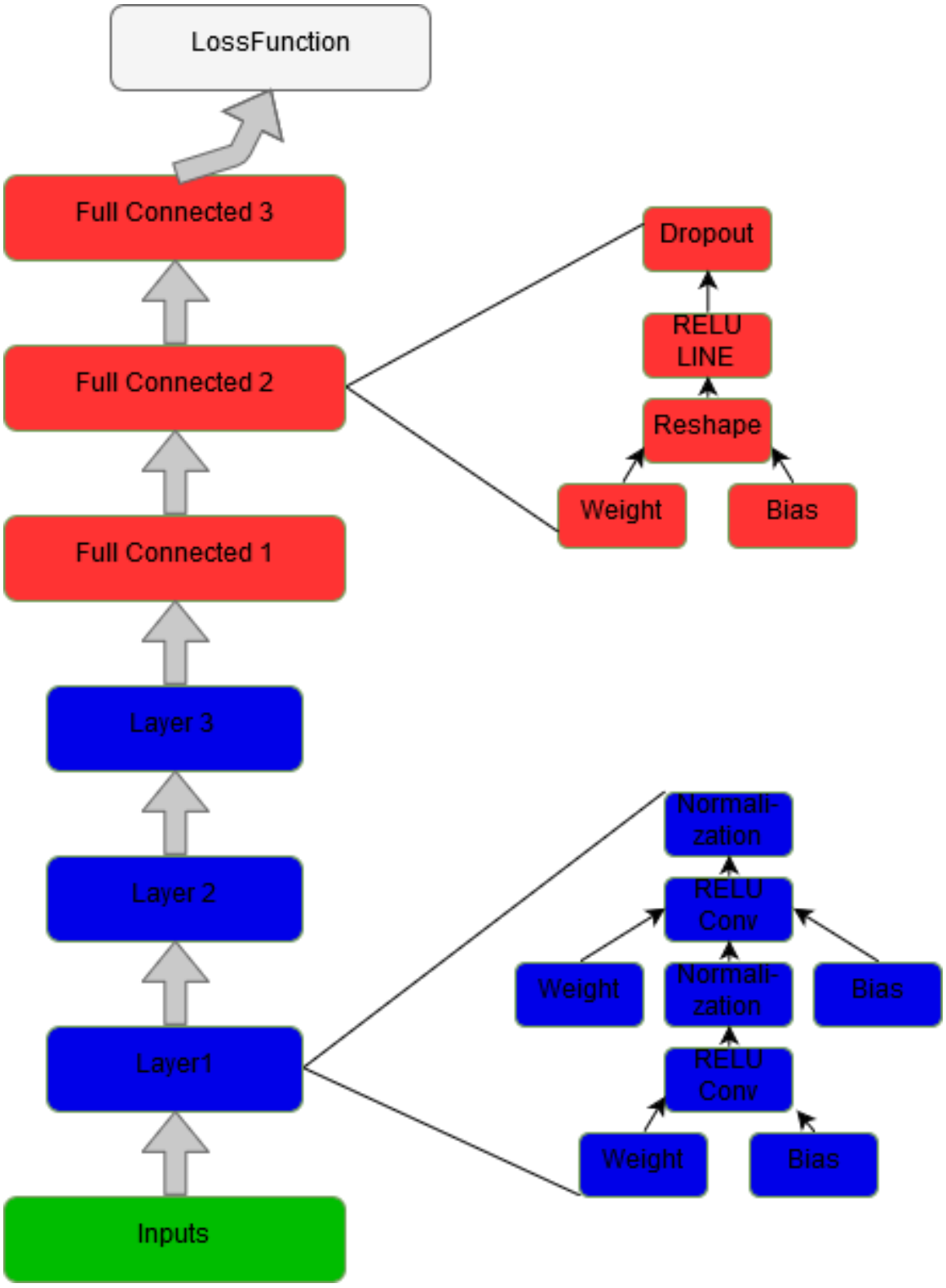

Figura 38: Grafo red neuronal local

4.2.2 Software para Azure

En este apartado se va a explicar detalladamente el funcionamiento del código fuente utilizado para las máquinas de Azure. El cual, aparte de su propósito de estudiar el rendimiento de Tensorflow con las distintas configuraciones posibles, ha sido

desarrollado de manera que realice un ajuste mediante odometría visual de la velocidad y los ángulos de inclinación de un vehículo, a partir de los datos obtenidos de un láser-scanner Lidar de 16 capas.

Los datos para la realización de estas pruebas han sido los de la biblioteca KITTI (14) car. Esta es una gran librería que contiene datos y funciones recogidos en muchísimas pruebas. Estas pruebas están realizadas con un vehículo, que lleva montados tantos sensores como le es posible. Algunos de los sensores son LIDAR de 32 y 16 capas, cámaras estéreo, GPS, radar, cámaras de propósito general, etc. En conclusión, es un conjunto muy completo de datos[11] públicos dedicados al análisis de la conducción y el desarrollo de vehículos autónomos.

El código fuente para esta parte del proyecto está compuesto por los siguientes ficheros:

- Training_and_test_graphic.py
- VO_KITTI_CNN_ALEX_graphic.py
- datagenerator.py
- Graficas.sh
- Environments (azure.py, local.py)

Esos códigos han sido desarrollados en Python 3.5, excepto el fichero gráficas.sh que ha sido implementado en bash. Este fichero es el encargado de guardar el estado de las tarjetas gráficas cada 30 segundos, con el objetivo de poder estudiar su comportamiento durante el entrenamiento posteriormente.

Dentro de "environments" se encuentran los ficheros azure.py y local.py estos ficheros contienen algunas variables necesarias para código principal. A continuación, se puede ver el contenido del fichero azure.py.

```
DATA_PATH = '/Data/ANN-PdE-LiDAR-SLAM-KITTI/data'

RESULTS_PATH = ' /ANN-PdE-LiDAR-SLAM-KITTI/results'

BASE_PATH = 'None'
```

El fichero datagenerator.py es un parser, encargado de traducir el contenido de los datos de entrada a unos datos legibles por el código fuente, por lo que no se explicara en detalle.

A continuación, se van a comentar las partes más importantes del código fuente, en concreto los ficheros Training_and_test_graphic.py y VO_KITTI_CNN_ALEX_graphic.py.

[11] http://www.cvlibs.net/datasets/kitti/eval_odometry.php

En primer lugar, el fichero VO_KITTI_CNN_ALEX_graphic.py contiene la definición de la arquitectura de la red neuronal convolucional.

```
def vo_kitti_cnn_alex_mod(x, keep_prob):

    """Create the network graph."""

    with tf.device('/device:GPU:0'):

        with tf.name_scope('Convolucional1'):

            out = conv(x, 10, 10, 32, 2, 2, padding='VALID',
name='conv1')

    with tf.device('/device:GPU:1'):

        with tf.name_scope('Convolucional2'):

            out = conv(out, 8, 8, 64, 2, 2, padding='VALID',
name='conv2')

        with tf.name_scope('Reshape'):

            out = tf.reshape(out, [-1,
np.prod(out.get_shape().as_list()[1:])])

    with tf.device('/device:GPU:2'):

        with tf.name_scope('FullConnected'):

            out = fc(out, out.shape[1], 512, name='fc1')

        with tf.name_scope('Dropout'):

            out = dropout(out, keep_prob)

    with tf.device('/device:GPU:3'):

        with tf.name_scope('FullConnectedOut'):

            out = fc(out, out.shape[1], 1, relu=False, name='fc2')

    return out
```

En el fragmento de código anterior, se encuentra la definición a más alto nivel de la arquitectura de la red neuronal convolucional. Para una visualización más sencilla de la arquitectura, se ha utilizado la herramienta Tensorboard para generar el grafo que se puede ver en la Figura 39: Grafo código Azure.

Aquí se puede observar que la arquitectura de la red está conformada con las siguientes capas:

1. Entradas
2. Convolucional
3. Convolucional
4. Reshape

5. Full Conected
6. Dropout
7. Full Connected

En el grafo están pintadas de diferentes colores dependiendo del dispositivo que la esté ejecutando, esto se puede observar en el código que define la arquitectura. Concretamente en la instrucción "tf.device", que le indica a Tensorflow a que dispositivo asignar cada función o ejecución, esto se hace con el objetivo de repartir la carga computacional entre los dispositivos disponibles. La instrucción "tf.name_scope" permite a Tensorboard identificar y agrupar un conjunto de funciones con el objetivo de que el grafo sea más claro y legible.

Ahora que ya se ha definido la arquitectura de la red se va a explicar las partes más importantes del código fuente principal, incluido en el fichero "Training_and_test_graphic.py".

En primer lugar, el código comienza recogiendo los argumentos de entorno que se han comentado anteriormente, tras ello se definen una serie de variables para guardar los resultados de la ejecución y se cargan los datos de entrada.

A continuación, se definen los "placeholder" para los datos de entrada, estos son puntos de conexión con Tensorflow utilizados para que este tenga acceso a datos externos a su ejecución. Se definen de la siguiente manera:

```
    with tf.device('/device:GPU:0'):
        with tf.name_scope('INPUTS'):
            # TF placeholder for graph input and output
            x = tf.placeholder(tf.float32, [batch_size, 28, 720, 2])
            y = tf.placeholder(tf.float32, [batch_size, num_classes])
            keep_prob = tf.placeholder(tf.float32)

            # learning rate
            lr = tf.placeholder(tf.float32)
```

Por último, en la fase de definición se carga la arquitectura de la red neuronal, así como la función de perdida, el optimizador y algunas variables adicionales.

```
out = vo_kitti_cnn_alex_mod(x, keep_prob)
loss_op = tf.sqrt(tf.reduce_mean(tf.square(out - y),
reduction_indices=0))
optimizer = tf.train.AdamOptimizer(learning_rate=lr)
```

```
train_op = optimizer.minimize(loss_op)

train_batches_per_epoch = int(np.floor(tr_data.data_size /
batch_size)) if batch_size else 1

val_batches_per_epoch = int(np.floor(val_data.data_size / batch_size))
if batch_size else 1
```

Una vez definido todo lo necesario se inicia la sesión de Tensorflow. Y comienza la fase de entrenamiento.

Dentro del entrenamiento hay 2 fases diferenciadas, el entrenamiento y la validación. El entrenamiento es el encargado de tratar de ajustar la red lo máximo posible para que ajuste las salidas calculadas con las reales. En la validación al contrario que en el entrenamiento la red no tiene acceso a los datos de las salidas reales por lo que debe calcular un resultado y este posteriormente se compara con la salida real para calcular la precisión de la red.

El entrenamiento esta codificado de la siguiente manera:

```
sess.run(train_op, feed_dict={x: img_batch_trn,
                                                y: label_batch_trn,
                                                keep_prob: dropout_rate,
                                                lr: learning_rate})
```

La validación esta codificado de la siguiente manera:

```
acc = sess.run(loss_op, feed_dict={x: img_batch,
                                                    y: label_batch,
                                                    keep_prob: 1.,
                                                    lr: learning_rate})
```

Estas dos instrucciones, lo que hacen es ejecutar las funciones train_op y loss_op de manera que se ejecuten sobre Tensorflow. Estas dos operaciones se ejecutan una serie de ciclos para ajustar la red al comportamiento esperado.

Por último, se guardan los resultados obtenidos y se cierra la sesión.

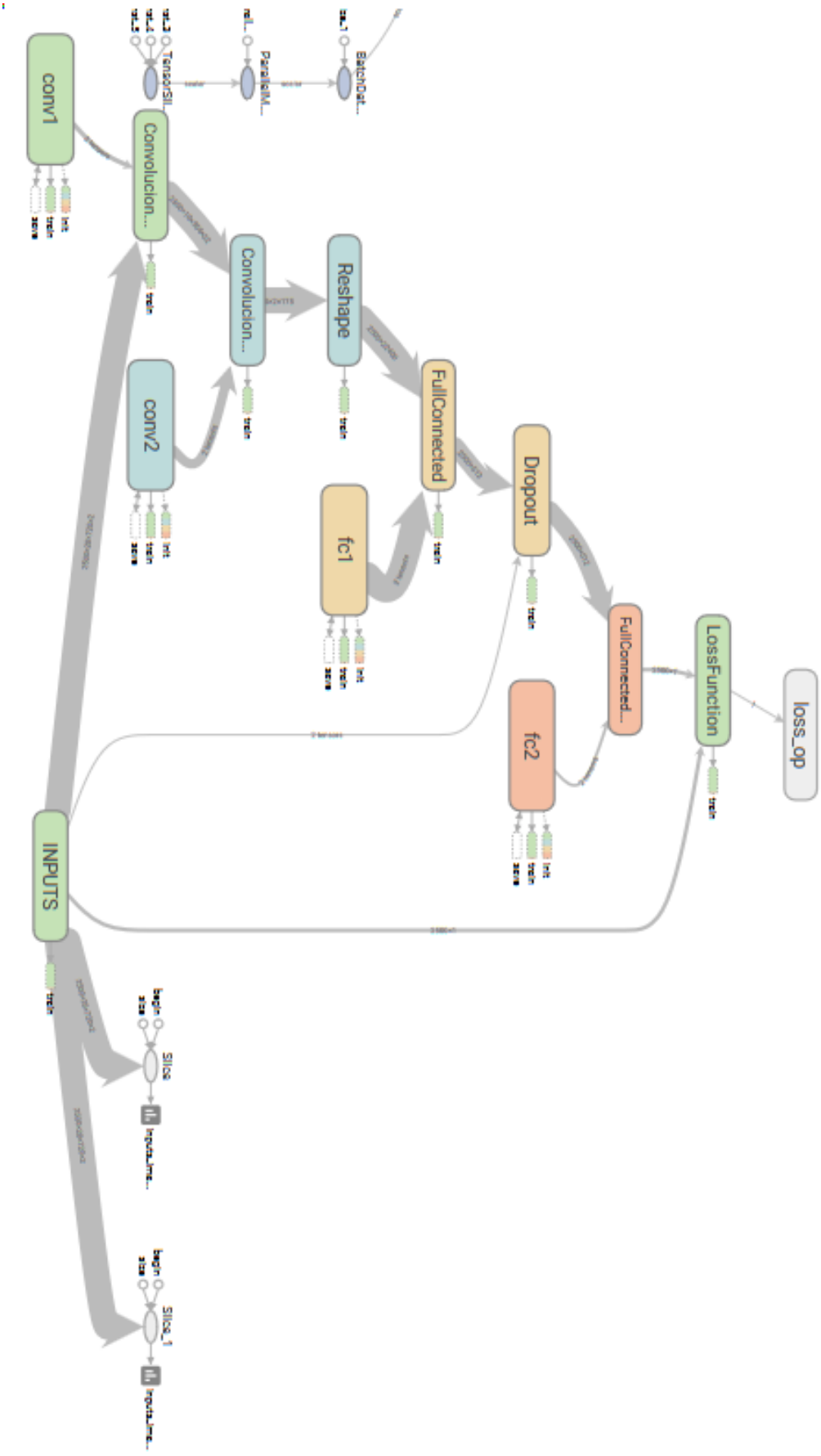

Figura 39: Grafo código Azure

Los resultados obtenidos en esta fase, en cuanto al ajuste de la red neuronal, del proyecto son los siguientes. Debido a que la precisión de la red neuronal se mide mediante la función de perdida, se va a exponer este mismo valor como resultado. Antes de nada, explicar que la función de pérdida es el computo del error obtenido del resultado que ha sido predicho y el resultado real. Esta función habitualmente viene dada por el error cuadrático medio o mediante validación cruzada.

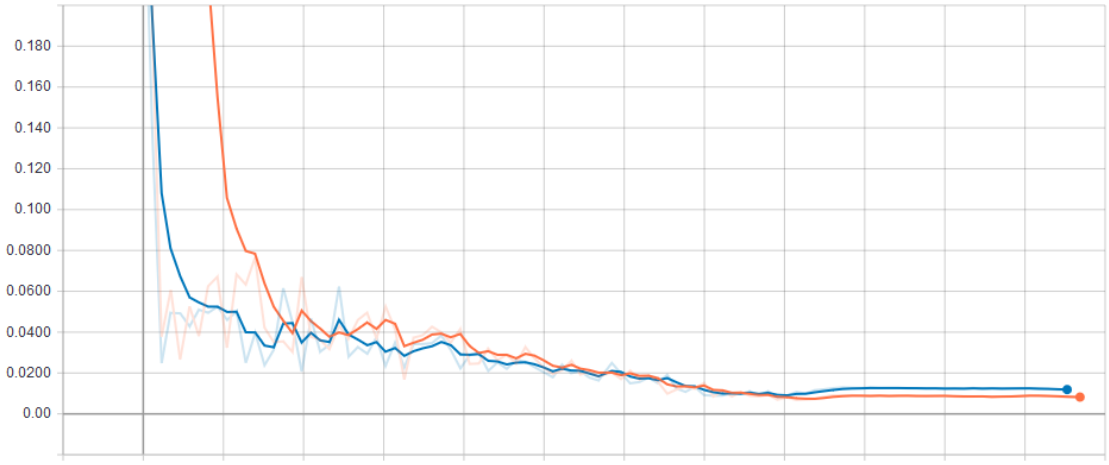

Figura 40: Función de perdida Azure

Como se puede observar en la Figura 40: Función de perdida Azure se ha conseguido una precisión importante por encima de un 99% de precisión o lo que es lo mismo un error inferior al 1%.

Otra conclusión que se ha extraído de esta fase del proyecto, más enfocada al aumento de rendimiento que proporciona una buena configuración de software es la siguiente. Se ha comprobado que, si la distribución de carga entre varias gráficas se realiza de manera automática, dejando que Tensorflow lo distribuya entre los dispositivos, el rendimiento es menor que si se realiza manualmente. Esto se debe a que cuando Tensorflow reparte la carga, trata de utilizar todos los recursos disponibles, en cuanto a tarjetas gráficas, pero solo utiliza una de ellas para las tareas de computación, del resto solo se utiliza la memoria disponible. Esto se puede comprobar aumentando el tamaño del batch, o conjunto de imágenes que se entrenan por ciclo, ya que lanza un error si se amplía lo suficiente como para que no quepa en una sola gráfica, en cambio sí se ha distribuido manualmente de manera adecuada, el tamaño del batch posible es mucho mayor.

Otro aspecto en el que afecta una buena distribución de la carga es en el tiempo por epoch, o tiempo se tarda en realizar un ciclo de aprendizaje. Ser ha comprobado que con una buena distribución de la carga computacional se reduce ligeramente el tiempo por epoch. Esto se ha comprobado asignando una capa a cada gráfica y comparándolo con la asignación de todas las capas a una gráfica. Los resultados numéricos obtenidos de esta forma son un tiempo por epoch de alrededor de 37 segundos para la carga

distribuida frente a los 45 segundos de la carga en una sola gráfica. Aun así, esta configuración no es la que da el mejor resultado ya que, aunque internamente cada capa trabaje de manera paralela, la forma de ejecución entre una capa y otra es secuencial. El mejor resultado lo dará una configuración que incluya capas que se ejecuten de manera paralela y que tengan una capa final que correlacione sus resultados. A la entrega de este proyecto este trabajo aún está en desarrollo.

4.3 Estudio de rendimiento Hardware

En este apartado se van a exponer los resultados del estudio de rendimiento de hardware que se ha realizado durante el proyecto. Este apartado, como el anterior se va a dividir en 2 fases.

Una primera fase en la que se han realizado pruebas en una maquina física disponible en el INSIA, en este apartado se mostrara el diferente comportamiento de las gráficas con las diferentes configuraciones de hardware que se pueden realizar en Tensorflow.

En la segunda fase se expondrá el trabajo realizado en Azure en donde se ha tratado de estudiar la manera de obtener la mejor configuración de hardware para obtener un máximo rendimiento de los recursos. De esta fase se extraerán las conclusiones de que tipo y configuración de hardware se debe buscar para obtener un rendimiento máximo del mismo, aprovechando al máximo los recursos disponibles.

4.3.1 Hardware utilizado

Hardware Local

Las especificaciones del sistema sobre el que se han utilizado para estas pruebas es el siguiente:

- Placa base: Gigabyte GA-Z270X-UD3
- Procesador: Intel Core i7-6700K 4GHz
- RAM: Kingston 16GB 2133MHz DDR4
- Grafica 1: NVIDIA Titan Xp 12GB
- Grafica 2: NVIDIA GeForce GTX 1080 Ti 8 GB
- Sistema operativo: Ubuntu 16.04 Xenial

Hardware Azure

Las especificaciones del sistema sobre el que se han utilizado para estas pruebas es el siguiente:

- Escenario virtualizado
- Procesador: 24 Núcleos
- RAM: 224GB
- Graficas: 4x NVIDIA Tesla K80
- Disco: 1440 GB SSD

- Sistema operativo: Ubuntu Server 16.04

4.3.2 Pruebas en la maquina local

Escenarios hardware

En este apartado se exponen los escenarios hardware que se han definido y utilizado en este trabajo. Como se ha mencionado anteriormente estos escenarios hardware se han definido mediante la configuración interna de sesión de TensorFlow con lo que se consigue que sin modificar el hardware físicamente, se pueda configurar como va a ser la ejecución del algoritmo.

En esta configuración de sesión se pueden definir multitud de parámetros configurables, mediante los cuales se modifica el comportamiento del algoritmo sobre el sistema. Algunos de estos parámetros configurables son indicar sobre que GPU y/o CPU se ejecuta el algoritmo, incluso se puede definir que parte de que algoritmo se ejecuta en cada dispositivo. También se puede realizar balanceo de carga manual entre los dispositivos, aunque Tensorflow ya realiza esta tarea automáticamente. Otro parámetro configurable es definir qué cantidad de memoria es utilizable en cada dispositivo, esto puede resultar útil si hay varios algoritmos ejecutándose a la vez o para configurar que no se puede utilizar toda la capacidad de memoria de las tarjetas gráficas, ya que si se utiliza el 100% de memoria los servicios de las tarjetas gráficas quedan bloqueadas.

4.3.3 Escenario 1(CPU)

En el primer escenario hardware se ha configurado Tensorflow para que funcione únicamente sobre CPU. Debido a esta configuración no se utiliza ni la capacidad de procesamiento ni la memoria disponible en las tarjetas gráficas. Únicamente se utiliza la capacidad de computación del procesador y la memoria RAM disponible.

Como se puede ver en las gráficas de las figuras desde Figura 41: Potencia consumida ambas gráficas hasta la Figura 43: Porcentaje de utilización ambas gráficas con esta configuración es 0, las gráficas no son utilizadas. No consumen potencia, nada más la necesaria para su funcionamiento básico y su memoria está en valores mínimos.

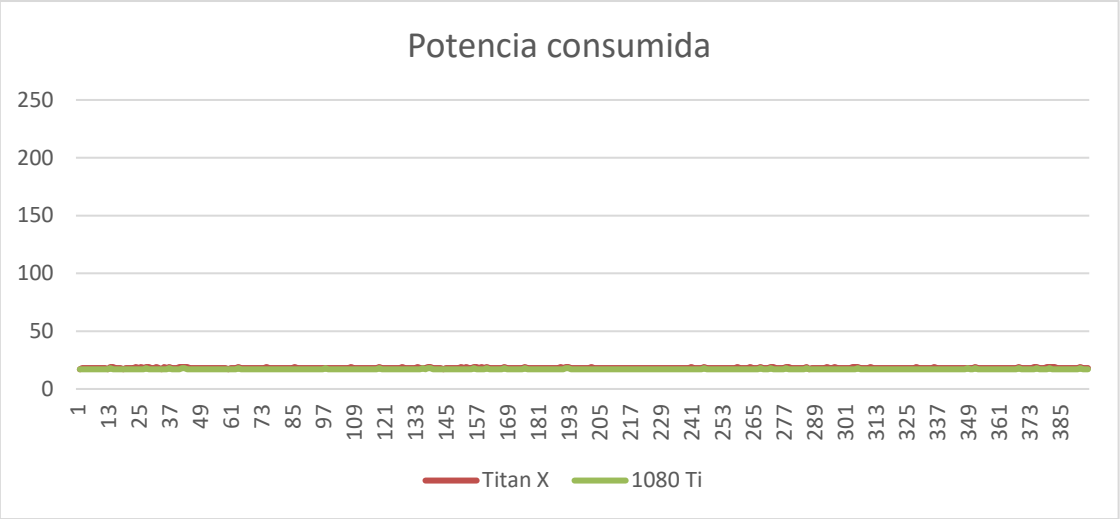

Figura 41: Potencia consumida ambas gráficas

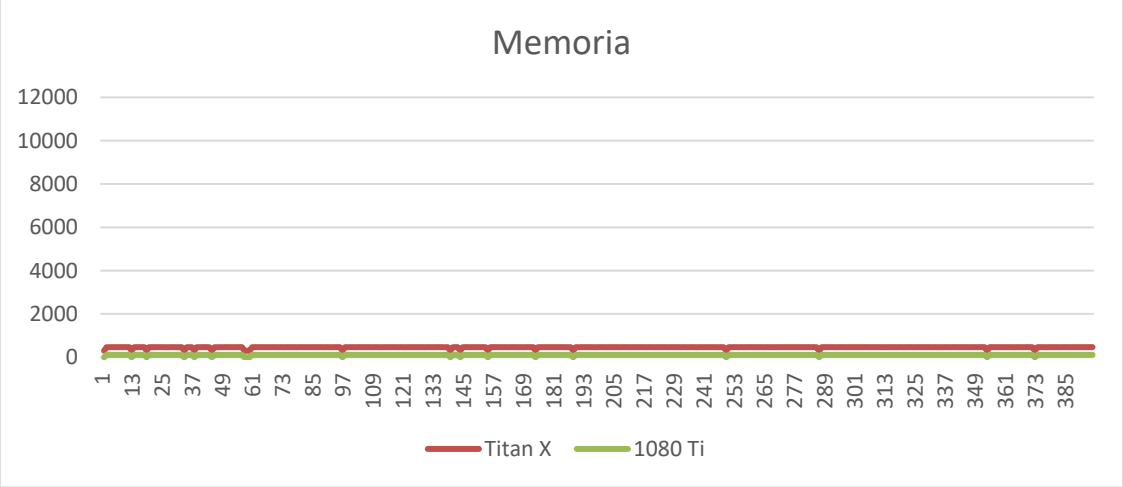

Figura 42: Memoria utilizada ambas gráficas

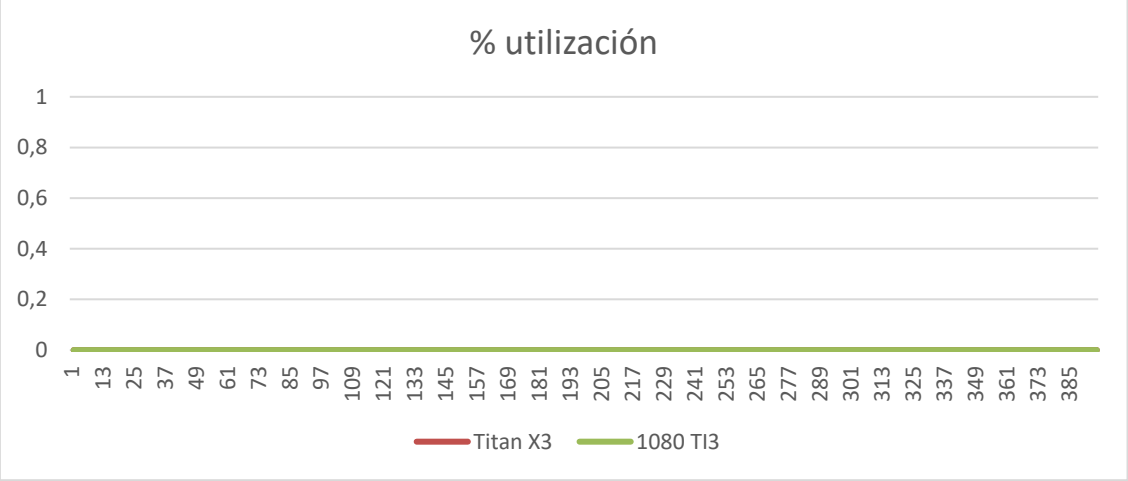

Figura 43: Porcentaje de utilización ambas gráficas

4.3.4 Escenario 2(2 graficas)

En el segundo escenario hardware se ha configurado Tensorflow para que se ejecute automáticamente en paralelo en las dos tarjetas gráficas disponibles. Para asegurar un rendimiento óptimo de las tarjetas gráficas se ha configurado adicionalmente que el algoritmo utilice como máximo un 90% de la memoria disponible en ambas tarjetas. Cabe destacar que debido a que las tarjetas utilizadas son diferentes no se puede hacer uso de un bus SLI entre las tarjetas lo que aumentaría un poco el rendimiento ya que con él se tiene un bus dedicado para la comunicación entre las tarjetas lo que permite no sobrecargar el bus PCI de la placa base.

Los resultados de la ejecución de esta configuración se pueden encontrar en las figuras desde la Figura 44: Potencia consumida ambas gráficas hasta la Figura 46: Porcentaje de utilización ambas gráficas. Esta configuración es mucho más interesante que la anterior, y es la más comúnmente utilizada. Con esta configuración Tensorflow se encarga, al menos en la teoría, de distribuir la carga de computo de manera paralela en las dos gráficas. Aunque en la práctica no es cierto.

Con una primera observación a las gráficas obtenidas se puede observar que ambas graficas están consumiendo potencia, con lo que se sabe que ambas están trabajando. Esto también se puede ver mediante el comando nvidia-smi donde se pueden ver 2 procesos Python cargados en las gráficas, uno por cada dispositivo. Esta conclusión sería similar al observar las gráficas de utilización de memoria o de utilización de la CPU.

Pero si se realiza una observación más profunda, concretamente en la Figura 46: Porcentaje de utilización ambas gráficas, aunque en la figura de incluida en este proyecto no se puede apreciar muy bien debido al tamaño, se puede observar que ambas gráficas nunca trabajan al mismo tiempo.

Lo único que hace Tensorflow es ciertamente repartir la carga computacional entre las dos gráficas, pero no paralelamente como dice la teoría y documentación[12] de Tensorflow sino de manera serializada, con lo que realmente no se consigue un aumento de potencia considerable en comparación con los escenarios 3 y 4 que se verán a continuación. Por supuesto, la capacidad de computo es muchísimo mayor que en la utilización de CPU del escenario 1, por muchos motivos como mayor y más rápida memoria principal, dedicación exclusiva a estos procesos o mayor capacidad computacional.

[12] https://www.tensorflow.org/programmers_guide/using_gpu

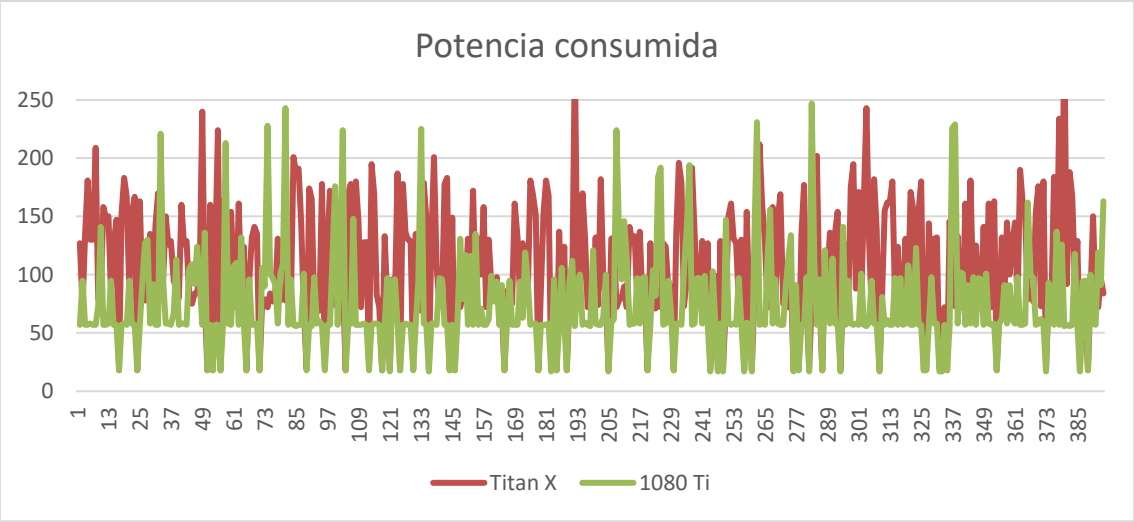

Figura 44: Potencia consumida ambas gráficas

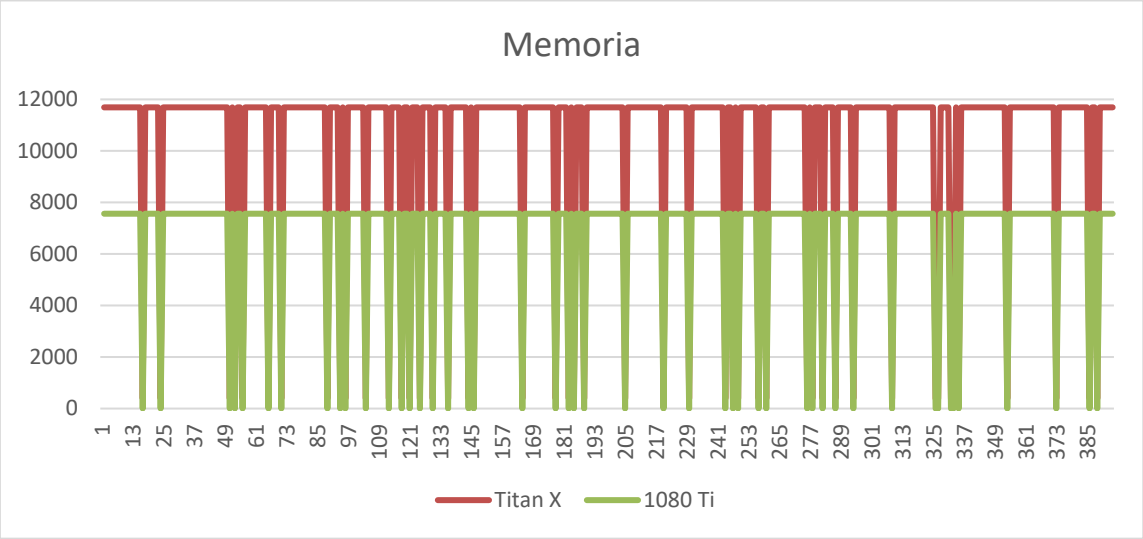

Figura 45: Memoria utilizada ambas gráficas

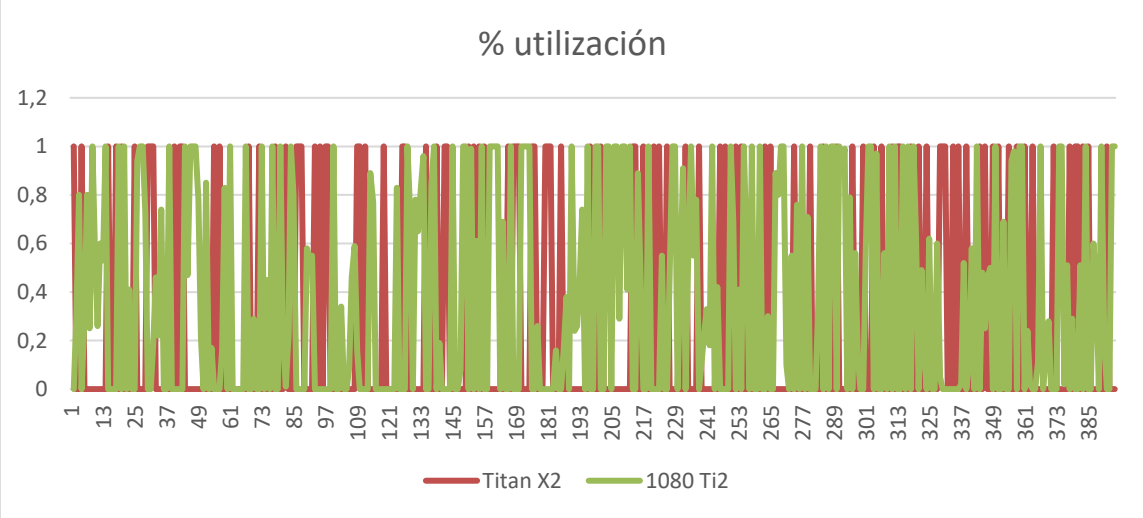

Figura 46: Porcentaje de utilización ambas gráficas

4.3.5 Escenarios 3 y 4

En los dos últimos escenarios probados se ha configurado Tensorflow para que utilice la capacidad de procesamiento de únicamente una de las dos graficas disponibles, aunque se ha comprobado que a pesar de que a la configuración indica que se use solo una de las dos graficas la memoria necesaria para Tensorflow se carga en ambas graficas a la vez, aunque el procesamiento solo se realiza en una. Los resultados de estas ejecuciones se pueden encontrar en las gráficas de la Figura 47: Potencia consumida ambas gráficas hasta la Figura 52: Porcentaje de utilización ambas gráficas.

En las figuras:

- Figura 47: Potencia consumida ambas gráficas
- Figura 50: Potencia consumida ambas gráficas

Y

- Figura 49: Porcentaje de utilización ambas gráficas
- Figura 52: Porcentaje de utilización ambas gráficas

Se puede ver que son complementarias las dos primeras, 47 y 50, en cuanto a potencia consumida, y las dos segundas, 49 y 52 en cuanto a la utilización. En cambio, si se observan las figuras:

- Figura 48: Memoria utilizada ambas gráficas
- Figura 51: Memoria utilizada ambas gráficas

Se puede ver que ambas gráficas tienen su memoria utilizada en un alto porcentaje, sobre el 90 % de su capacidad, como se configuró en Tensorflow, esto se debe a que si Tensorflow detecta que no puede incluir todos los datos utilizados en sus correspondientes tamaños de batch utilizados, utiliza todos los recursos disponibles en cuanto a memoria, pero respeta la configuración de utilización que se ha realizado.

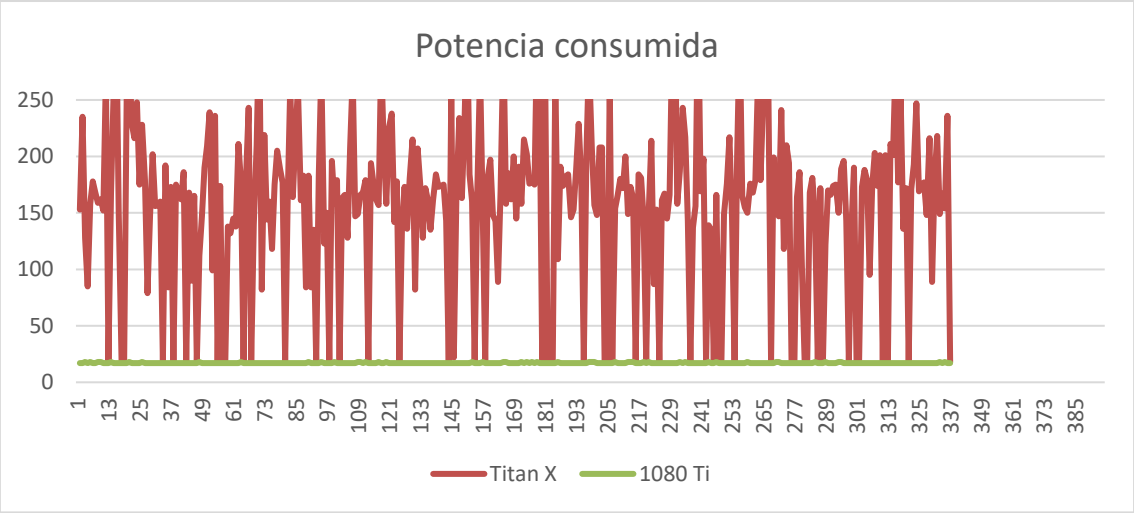

Figura 47: Potencia consumida ambas gráficas

Figura 48: Memoria utilizada ambas gráficas

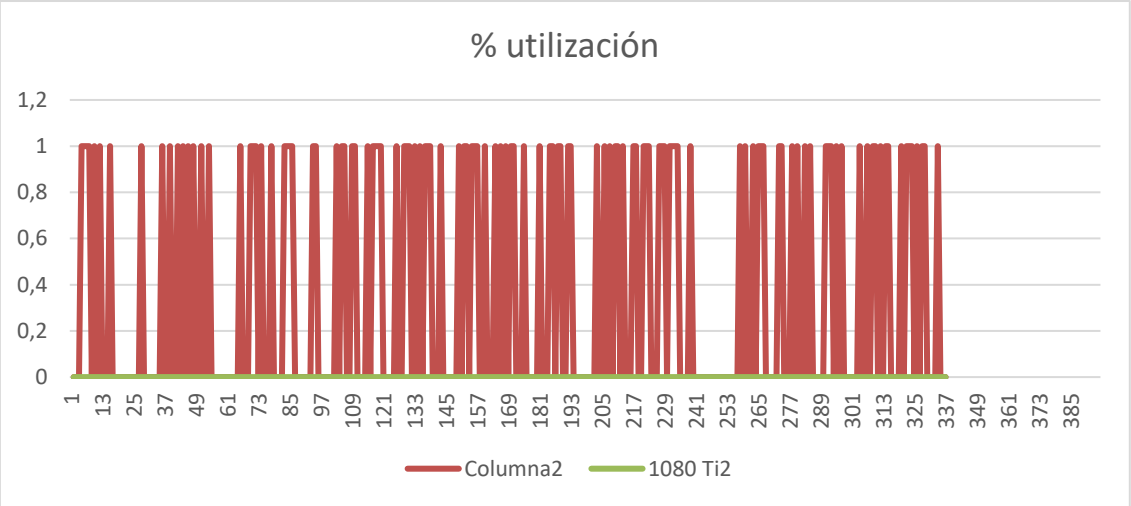

Figura 49: Porcentaje de utilización ambas gráficas

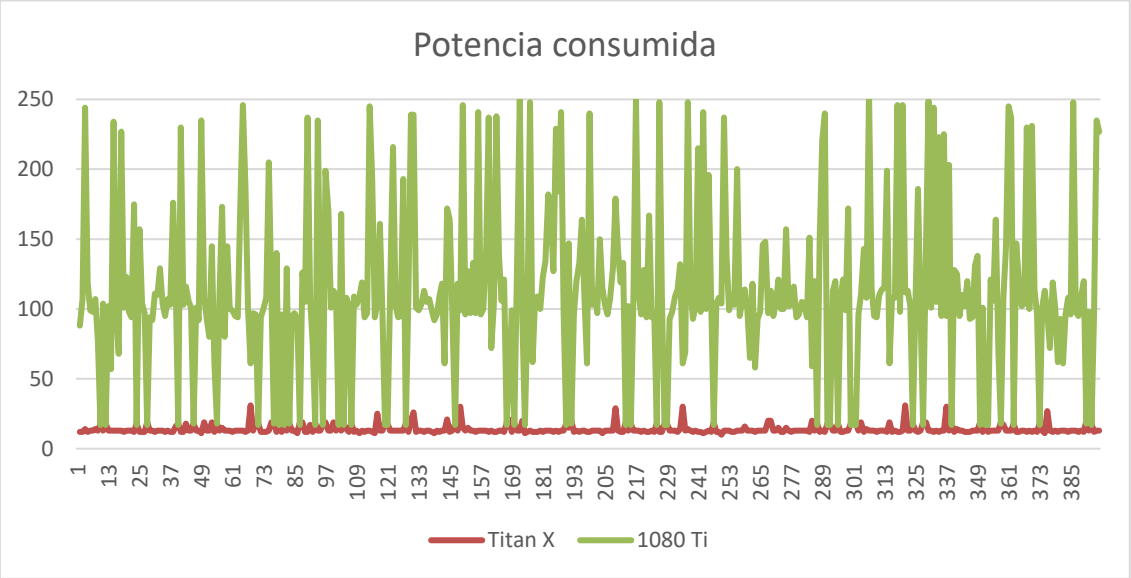

Figura 50: Potencia consumida ambas gráficas

Figura 51: Memoria utilizada ambas gráficas

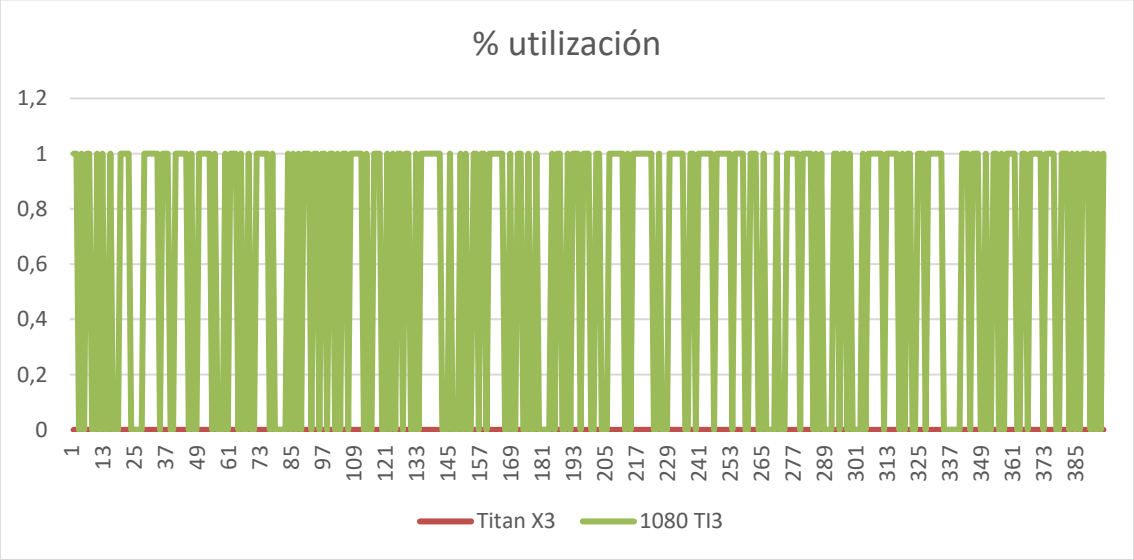

Figura 52: Porcentaje de utilización ambas gráficas

Las conclusiones que se han extraído de estos experimentos es que la configuración mediante el framework de Tensorflow de la utilización de los dispositivos, ciertamente se refleja en estos, pero solo en concepto de computación, en el caso de la memoria, toda la memoria disponible es utilizada si es necesario.

En cuanto a velocidad de computación, se puede asegurar que la utilización de las GPUs proporciona un aumento de velocidad importante respecto a la computación sobre la CPU. Dentro de la computación mediante GPUs se obtiene que se consigue un pequeño aumento de la velocidad en el caso de permitir a Tensorflow distribuir la carga entre las gráficas automáticamente, aunque no es significativo. Se consigue un aumento significativo de la velocidad de computación mediante la repartición de la carga

computacional mediante una distribución óptima de la arquitectura de la red neuronal convolucional entre los dispositivos disponibles.

4.3.6 Pruebas en la maquina Azure

En este apartado se van a exponer las pruebas que se han realizado en Azure. Partiendo de todas las pruebas sobre el hardware que se realizaron en el apartado anterior, en este apartado se ha tratado de desarrollar mucho más la parte software, aun así, se han realizado dos pruebas importantes que muestran la diferencia de comportamiento de las tarjetas gráficas cuando se trata de ejecutar la red neuronal en todas las gráficas en paralelo o en el caso de que se ejecute todo sobre una misma gráfica.

En estas dos pruebas, se ha utilizado la misma técnica que en apartado anterior, mediante la modificación de la configuración de hardware a través de Tensorflow.

En la primera prueba se forzado la ejecución de todas las capas de la red neuronal convolucional sobre la misma tarjeta gráfica. En el código, concretamente en la definición de la arquitectura de la red neuronal se ha indicado a tensorflow que ejecute cada capa en la gráfica 0 mediante el siguiente comando:

```
with tf.device('/device:GPU:0'):
```

Con ello se consigue que toda la carga computacional de esta capa sea realizada por la gráfica 0.

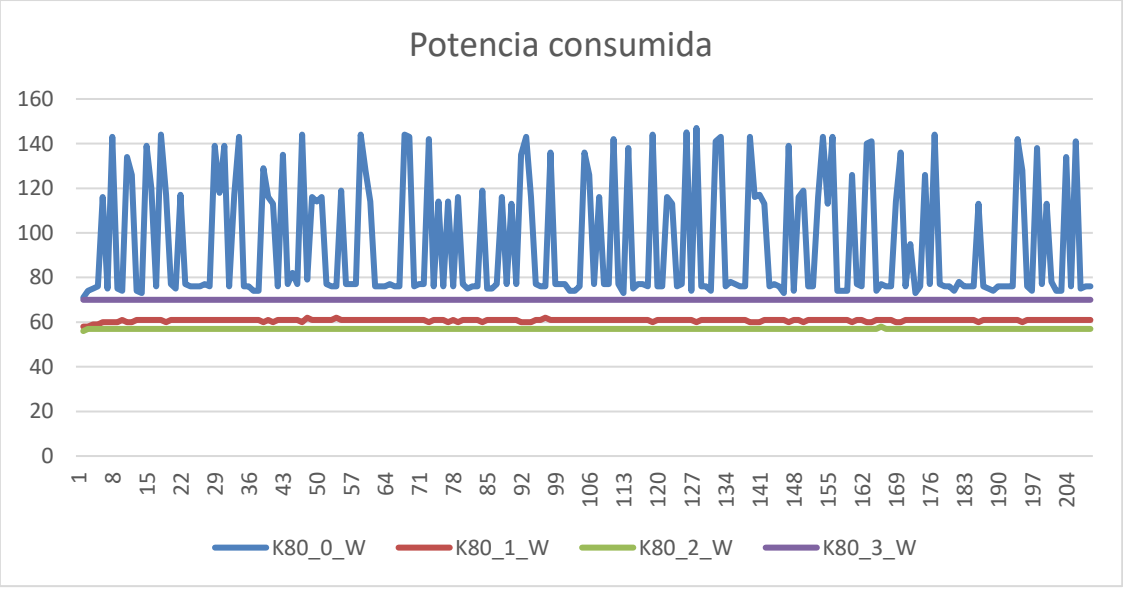

Figura 53: Potencia consumida

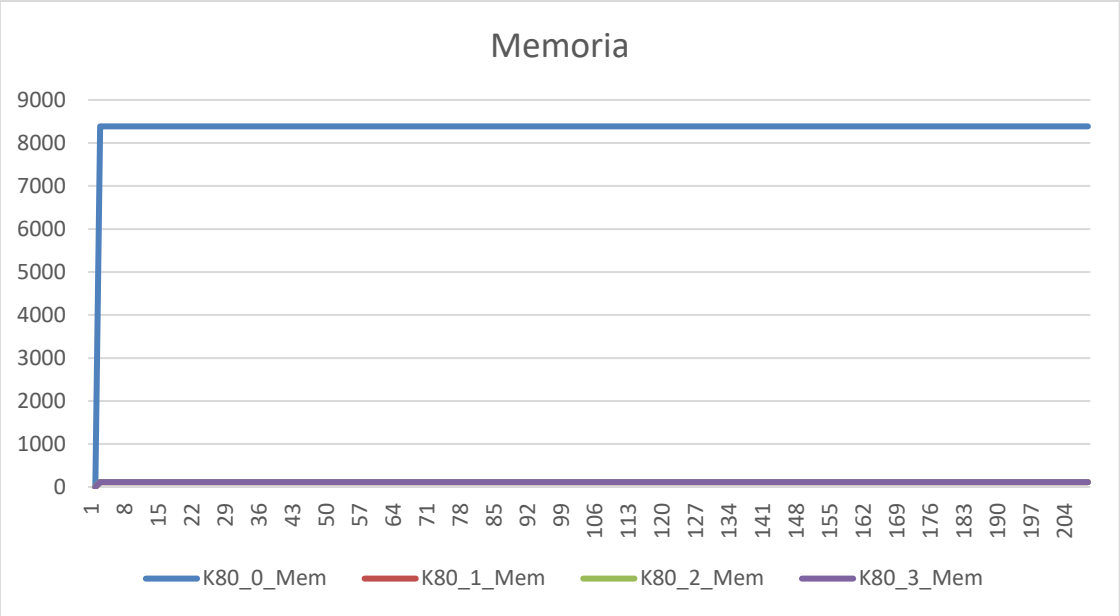

Figura 54: Memoria utilizada

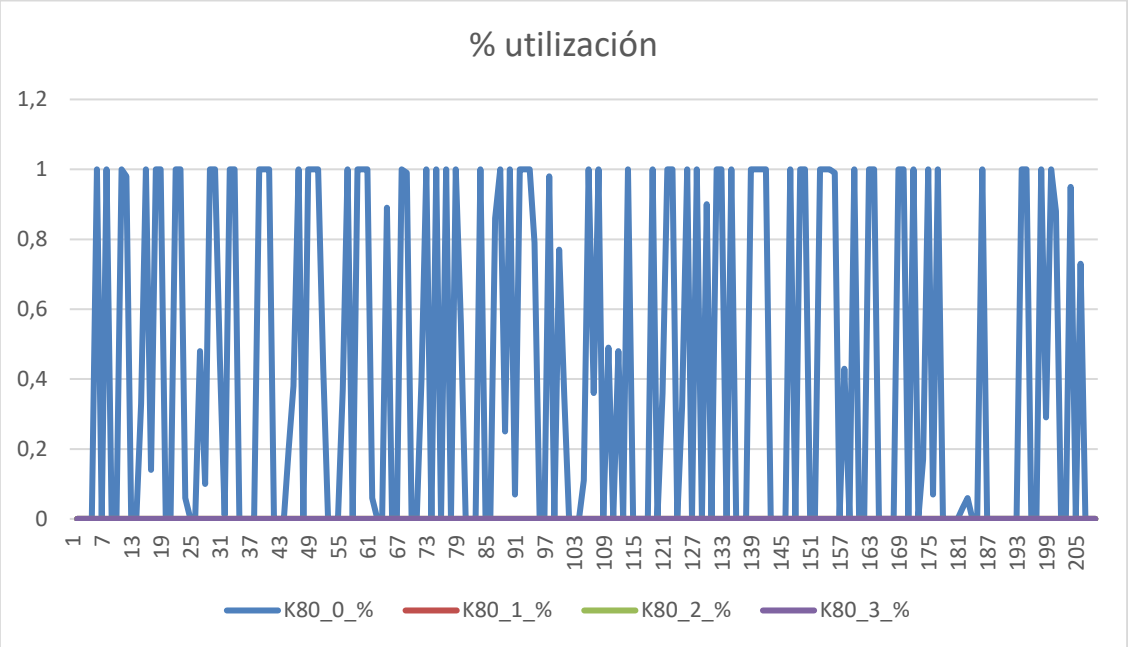

Figura 55: Utilización

Como se puede observar en las figuras:

- Figura 53: Potencia consumida
- Figura 54: Memoria utilizada
- Figura 55: Utilización

La gráfica 0, pintada de color azul en todas las figuras, está soportando toda la potencia computacional. Esto se puede observar sobre todo en la Figura 53: Potencia consumida ya que se ve que la única de las 4 gráficas que está consumiendo potencia es la 0. En este experimento, debido al tamaño del batch bien ajustado y al tamaño de los

datos de entrada la única gráfica con memoria utilizada es la 0 pero Tensorflow cuenta con una función que, aunque se fuerce la utilización de la gráfica 0, si el tamaño del conjunto de datos es demasiado grande o el batch está mal ajustado a la memoria disponible, Tensorflow utilizará la memoria libre de los otros dispositivos.

La Figura 55: Utilización también presenta de forma clara las gráficas se están utilizando ya que, aunque están dibujados los gráficos de utilización de las 4 gráficas, la única en la que se puede apreciar utilización es la 0.

La segunda prueba que se ha realizado ha consistido en tratar de repartir la carga computacional entre las gráficas responsabilizando a cada gráfica disponible la computación de una de las capas de la red neuronal convolucional, tal como se explicó en el apartado de software.

Al realizar el experimento, el comportamiento de las gráficas obtenido ha sido el siguiente:

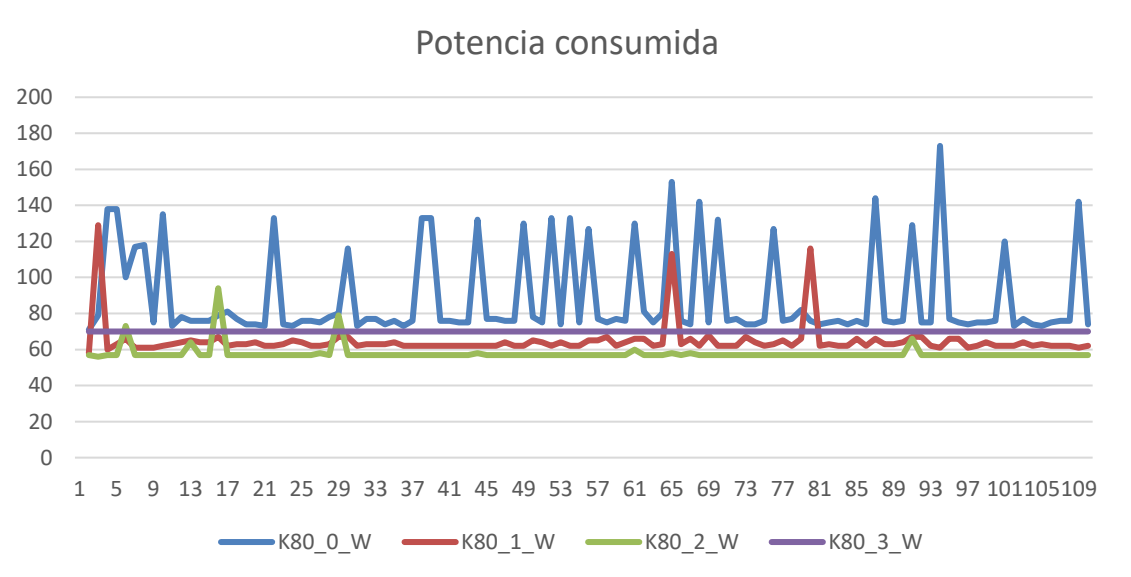

Figura 56: Potencia consumida

Figura 57: Memoria utilizada

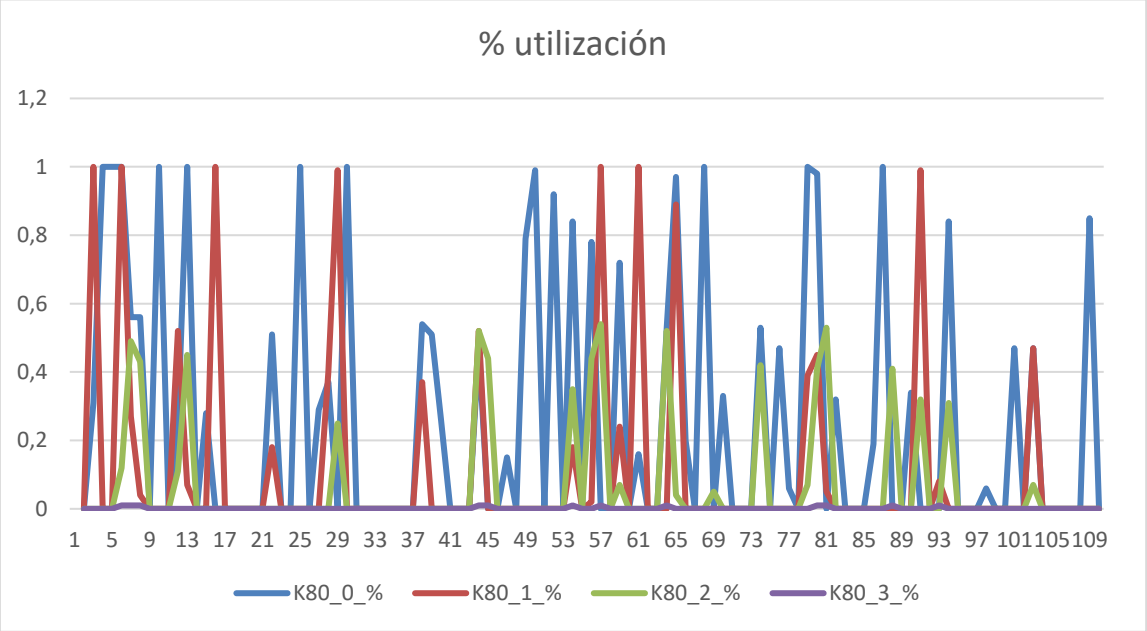

Figura 58: Utilización

Como se puede observar en la graficas anteriores, en este caso a diferencia del primer experimento todas las gráficas están siendo utilizadas. Se puede ver claramente en la Figura 58: Utilización ya que se ve como se superpone la computación paralela de las 4 gráficas. Cabe destacar que en la Figura 57: Memoria utilizada se puede observar que la gráfica 0 tiene mucha más memoria utilizada que el resto, esto se debe a que en la gráfica 0 se encuentran los placeholder, o puntos de entrada a Tensorflow, por lo que todos los datos de entrada pasan por ella.

Las conclusiones principales, en cuanto a hardware que se han extraído a partir de los experimentos realizados en Azure son que la característica más importante para que la ejecución sea lo más rápida posible, es que la gráfica que realice la computación tenga una gran potencia de computación. Se ha podido comprobar que la cantidad de memoria disponible pasa a un segundo plano de importancia, ya que Tensorflow se apoyara en los recursos disponibles utilizando memoria de otras gráficas.

También se ha podido comprobar que resulta más eficiente tener una solo tarjeta gráfica más potente que varias tarjetas gráficas menos potentes, ya que a no ser que se aplique técnicas novedosas de clustering o repartición de carga entre las gráficas, comenzando por adaptar la arquitectura de la red, no se conseguirá un aumento de rendimiento importante.

5 Conclusiones

Las conclusiones que se han extraído de la realización de este proyecto han sido las siguientes.

En cuanto al hardware las conclusiones que se han realizado es que la elección arquitectura optima del hardware a utilizar se puede definir de dos maneras dependiendo de los recursos que se quieran emplear. En el caso de que los recursos sean limitados como para adquirir equipos con varias tarjetas gráficas, resulta más eficiente comprar una sola tarjeta gráfica de mayor potencia que varias de menor potencia. Esto se debe a que las tareas de paralelización de computo mediante software son bastante complejas y resulta más rentable ejecutar todo sobre una misma gráfica. En el caso de que se cuente con mayor capital a invertir, la opción más eficiente será adquirir tarjetas gráficas potentes configuradas mediante los métodos expuestos en este trabajo.

En cuanto al software la solución que proporciona mayor rendimiento es la paralelización. Como se ha expuesto en este trabajo una paralelización por capas de la red neuronal optimiza el uso de memoria, pero no aporta un rendimiento significativo a la computación. Por otro lado, una paralelización mediante grafos separados en cada tarjeta gráfica y unidos en su capa final por una capa full connected que proporcione una salida no lineal, proporcionará un aumento de rendimiento exponencial.

Como conclusión, se asegura que la solución con mayor rendimiento conjunto entre software y hardware será la implementada mediante un clúster de Tensorflow, el cual proporciona tanto paralelización software como hardware, lo que permitirá un aumento de rendimiento importante. Este es la línea de trabajo por donde seguirá el proyecto a la finalización de esta tesis.

6 Líneas de trabajo futuro

Las principales líneas de trabajo futuro previstas para este proyecto son 2, las cuales están muy diferenciadas ya que una va destinada al software y otra al hardware.

La primera línea de trabajo futuro es mejorar la optimización de la red neuronal de manera que se solucionen los problemas de overfitting, underfitting, etc. Para ello hay distintas configuraciones que se deben ajustar.

En primer lugar, la arquitectura de la red, esta debe ser coherente con los datos disponibles, una arquitectura muy grande o pequeña, tanto en las capas como en profundidad o número de capas puede ser perjudicial para el aprendizaje.

Por otro lado, se deben optimizar ajustes como el learning_rate adaptativo en el tiempo, el tamaño del batch, el número de iteraciones, etc.

Todas estas configuraciones, y muchas otras deben ser ajustadas al unísono ya que la diferente combinación de ellas afectara al comportamiento y resultado del aprendizaje.

La segunda línea de trabajo futuro es la implementación de un clúster de Tensorflow, el cual es capaz de aprovechar el rendimiento individual de varias máquinas para conseguir un rendimiento conjunto mayor con el objetivo de acelerar el aprendizaje.

Esta línea de trabajo tiene previsiones a medio-largo plazo ya que se trata de un campo complejo, en el que se utiliza tecnología que está en pleno desarrollo y por lo tanto tiene fallos, poca comunidad detrás que haya trabajado en ello, además de requerir un esfuerzo monetario importante ya que cada una de las maquinas utilizadas debe cumplir unos requisitos técnicos de capacidad computacional importante, lo que no es barato.

7 Bibliografía

1. **Lidar velodyne vlp-16. [En línea] 10 de 11 de 2017. http://velodynelidar.com/vlp-16.html.**

2. Tensorflow. Pagina oficial Tensorflow. [En línea] 22 de 02 de 2018. https://www.tensorflow.org/.

3. —. Herramienta de pruebas de redes neuronales con Tensorflow. [En línea] 22 de Marzo de 2018. http://playground.tensorflow.org.

4. INSIA. [En línea] 01 de 01 de 2018. http://insia-upm.es/.

5. Azure. Recursos azure. [En línea] 15 de 03 de 2018. https://azure.microsoft.com/es-es/resources/.

6. F. Garrido, D. González, V. Milanés, J. Pérez, F. Nashashibi. Real-time planning for adjacent consecutive intersections. ***2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC), Rio de Janeiro, 2016, pp. 1108-1113.*** **[En línea] http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7795695&isnumber=7795515.**

7. Rahul Kala, Kevin Warwick. Reactive Planning of Autonomous Vehicles for Traffic Scenarios. [En línea] https://www.researchgate.net/publication/283860527_Reactive_Planning_of_Autonomous_Vehicles_for_Traffic_Scenarios.

8. Noor Cholis Basjaruddin, Kuspriyanto Kuspriyanto, Didin Saefudin, Edi Rakhman, Adin Mochammad Ramadla. Overtaking Assistant System Based on Fuzzy Logic. [En línea] http://journal.uad.ac.id/index.php/TELKOMNIKA/article/view/499.

9. NVIDIA. Guia de instalacion de cuda 9.0. [En línea] 8 de Marzo de 2018. http://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html.

10. —. Guia de instalación cuDNN. [En línea] 8 de Marzo de 2017. http://docs.nvidia.com/deeplearning/sdk/cudnn-install/index.html.

11. —. Pagina de descarga CUDA. [En línea] 8 de Marzo de 2018. https://developer.nvidia.com/cuda-90-download-archive?target_os=Linux&target_arch=x86_64&target_distro=Ubuntu&target_version=1604&target_type=deblocal.

12. —. Pagina de descarga cuDNN. [En línea] 8 de Marzo de 2018. https://developer.nvidia.com/rdp/cudnn-download.

13. Tensorboard. [En línea] 25 de 01 de 2018. https://www.tensorflow.org/programmers_guide/summaries_and_tensorboard.

14. KITTI dataset odometry. [En línea] 15 de 1 de 2018. http://www.cvlibs.net/datasets/kitti/eval_odometry.php.

15. Barrio, Alfredo Valle. ***Segmentación de nubes de puntos y fusión de datos en el ámbito de los vehículos autónomos.*** **Madrid : UPM, 2017.**

16. ***In-Datacenter Performance Analysis of a Tensor Processing Unit.*** **Jeremy, Norman P. Jouppi and Cliff Young and Nishant Patil and David Patterson and Gaurav Agrawal and Raminder Bajwa and Sarah Bates and Suresh Bhatia and Nan Boden and Al Borchers and Rick Boyle and Pierre-luc Cantin and Clifford Chao and Chris Clark and. 2017.**

17. ***Application of Deep Learning to Route Odometry Estimation from LiDAR Data.*** **Clavijo, M., Serradilla, F., Naranjo, J. E., Jiménez, F., Díaz, A. VEHICULAR 2017, The Sixth International Conference on Advances in Vehicular Systems, Technologies and Applications : s.n., 2017.**

18. ***Deep Learning Application for 3D LiDAR Odometry Estimation in Autonomous Vehicles.*** **Clavijo, M., Díaz, A., Serradilla, F., Jiménez, F., Naranjo, J. E. 7th European Transport Research Conference. TRA 2018 Viena (Austria) : s.n., 2018.**

19. ***Modelling the human lane-change execution behaviour through Multilayer Perceptrons and Convolutional Neural Networks.*** **Alberto Díaz Álvarez, Miguel Clavijo, Felipe Jiménez, Edgar Talavera, FranciscoSerradilla. 134-148, Transportation Research Part F: Traffic Psychology and Behaviour : s.n., 2018, Vol. 56.**