

CloudNAO: Una arquitectura de software para la integración de cómputo en la nube con robots NAO

Ivan Raymundo Feliciano Avelino

25 de julio de 2018

Contenido

Introducción	3
1. Antecedentes	4
1.1. NAOqi: Un marco de trabajo para desarrollar sobre robots NAO	4
1.1.1. Robot NAO	4
1.1.2. NAOqi	5
1.2. Algoritmos de aprendizaje profundo y TensorFlow	20
1.2.1. Aprendizaje automático	20
1.2.2. Redes neuronales artificiales	29
1.2.3. Aprendizaje profundo	41
1.2.4. Redes neuronales convolucionales	43
1.2.5. TensorFlow	46
1.3. Cómputo en la nube	52
1.3.1. Capas del cómputo en la nube	52
1.3.2. Servicios en la nube	54
1.3.3. Tipos de cómputo en la nube	55
1.3.4. Cómputo en la nube vs servicios en la nube	55
1.3.5. Tecnologías que hacen posible el cómputo en la nube	57
1.3.6. Características del cómputo en la nube	58
1.3.7. Robótica en la nube	58
1.4. Arquitecturas de Servicios de Transferencia de Estado Representacional	59
1.4.1. Protocolo de Transferencia de Hipertexto	59
1.4.2. World Wide Web	63
1.4.3. Transferencia de Estado Representacional (REST)	65
1.5. Servicios de BaaS y Firebase	67
1.5.1. Backend as a Service	67
1.5.2. Firebase	67
1.6. Interfaces de Programación de Aplicaciones de Transferencia de Es- tado Representacional	71
1.6.1. API de Google Cloud Vision	71
1.6.2. API de Kairos	74

1.6.3. API de Wit.ai	76
1.6.4. API REST de Firebase Database	78
2. La arquitectura CloudNAO	79
2.1. Descripción de la arquitectura CloudNAO	79
2.1.1. Servidor LAR	81
2.1.2. API REST de CloudNAO	81
2.1.3. Firebase Realtime Database y CloudNAO	109
2.1.4. Aplicación para Android CloudNAO	114
2.1.5. Firebase y el robot NAO	139
2.1.6. Aplicación web de CloudNAO	144
2.2. Implementación de casos de estudio con servicios en la nube existente	159
3. Un servicio de cómputo en la nube sobre la arquitectura CloudNAO	160
3.1. Descripción del servicio sobre CloudNAO	161
3.1.1. Descripción del conjunto de datos	161
3.1.2. Arquitectura de la CNN	162
3.2. Implementación del servicio sobre CloudNAO	165
3.2.1. Hardware y software	166
3.2.2. Preparación del conjunto de datos	166
3.2.3. Construcción del modelo en TensorFlow	167
3.2.4. Resultados	170
Conclusiones	174
Anexos	175
Bibliografía	175

Introducción

Actualmente existen diferentes plataformas que ofrecen servicios de cómputo en la nube para solucionar problemas de visión computacional o procesamiento de lenguaje natural usando modelos de aprendizaje profundo; también incluyen servicios para crear, entrenar y lanzar modelos creados por los desarrolladores. Todos estos recursos se entregan a través de interfaces que permiten que casi cualquier dispositivo con acceso a internet pueda acceder a éstos. El cómputo en la nube es un estilo de cómputo que permite el acceso a recursos informáticos a través de internet cuando un usuario lo demanda. El aprendizaje profundo es un tipo de aprendizaje automático donde las computadoras construyen conceptos complejos a partir de conceptos más simples.

NAO es un robot humanoide programable y autónomo que a pesar de contar con un poder de procesamiento suficiente para realizar tareas como jugar un partido de fútbol con otros robots, la aplicaciones modernas exigen que los dispositivos se adapten constantemente a nuevas necesidades de los usuarios, como el acceso a los datos del robot desde cualquier lugar, la clasificación de imágenes y detección de objetos en múltiples categorías, el procesamiento de lenguaje natural, etc.

El objetivo de esta investigación es desarrollar e implementar un sistema que permita la integración de servicios en la nube con robots NAO. Los recursos son brindados por las plataformas Google Cloud, Wit.ai, Kairos, Firebase y por el mismo Laboratorio de Algoritmos para la Robótica. Los servicios en la nube permitirán al robot NAO realizar tareas como el procesamiento de lenguaje natural de un discurso oral, el procesamiento de imágenes para reconocimiento de rostros y objetos, la clasificación de escenarios en una imagen, la conexión desde diferentes dispositivos con el robot y el acceso a datos generados por éste.

Capítulo 1

Antecedentes

1.1. NAOqi: Un marco de trabajo para desarrollar sobre robots NAO

1.1.1. Robot NAO

NAO es un robot humanoide autónomo y programable desarrollado por la empresa de Aldebaran Robotics.

El robot NAO mide 57.3 cm de altura, 27.3 cm de ancho, y pesa menos de 4.3 kg. El cuerpo está construido de un material de plástico y tiene una batería de ion de litio que lo abastece para un uso normal de aproximadamente 90 minutos o 60 en un modo activo.

Las versiones *V5* y *V4* tiene una CPU Atom de 1.6 GHz, 1 GB de RAM, una memoria flash de 2 GB y una micro SDHC de 8GB.

Se comunica remotamente con otros dispositivos mediante WiFi o por medio de un cable Ethernet. También cuenta con un puerto USB cuyo principal uso es para añadir un dispositivo como un sensor 3D o un Arduino.

Para la parte multimedia, el robot está equipado con un sistema de transmisión estéreo compuesto por dos bocinas en las orejas. Dos micrófonos en la cabeza con un paso de banda eléctrico entre 300Hz y 8kHz. Dos cámaras idénticas están localizadas en la parte anterior de la cabeza. Éstas proveen imágenes con una resolución de hasta 1280px × 960px y 30 cuadros por segundo.

Cuenta con diodos emisores de luz distribuidos entre la cabeza, orejas, ojos, pecho y pies. Cada uno de estos últimos cuenta con resistencias sensibles a la fuerza, que

son sensores encargados de medir la resistencia al cambio de acuerdo a una presión aplicada. Trabajan en un rango de 0 N a 25 N.

En el torso está localizada una unidad de medición inercial, compuesta por un girómetro y un acelerómetro, la cual permite una estimación de la velocidad y postura del torso.

El robot está equipado con dos sensores ultrasónicos, que sirven para estimar la distancia a obstáculos. Dependiendo de la versión del robot, el rango de detección varía de 0.20 cm a 0.80 cm en la última versión y 0.25 cm a 2.55 cm en versiones anteriores. Cuando la distancia es menor al límite inferior de cada versión, el robot únicamente sabe que hay un obstáculo presente. Tiene un cono efectivo de 60°.

Entre los sensores restantes están los de posición de las articulaciones, los de contacto y táctiles. Los dos últimos están en la cabeza, manos, pecho y pies.

Tiene en total 25 articulaciones repartidas entre la cabeza, brazos, piernas y pelvis. Todas las articulaciones cuentan con controladores de posición. Dada una articulación que enlaza dos partes del cuerpo del robot, la parte del cuerpo que está más cerca del tronco se considera fija y la parte que está más lejos es la que rota alrededor de los ejes de la articulación. Para realizar una rotación de las partes del cuerpo, definimos un sistema de referencia en cada articulación. En el sistema de referencia se tiene tres ángulos de rotación: *roll* (dirección), *pitch* (elevación) y *yaw* (alabeo). Las rotaciones *roll*, son alrededor del eje X, las *pitch* sobre Y y *yaw* con respecto a Z. La Figura 1.1 muestra el sistema de referencia.

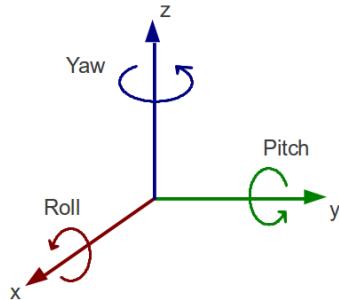


Figura 1.1: Sistema de referencia de los ángulos de rotación

En la Figura 1.2 se muestran algunos de los componentes del robot NAO.

1.1.2. NAOqi

NAOqi es el nombre del software principal que corre sobre el robot y lo controla.

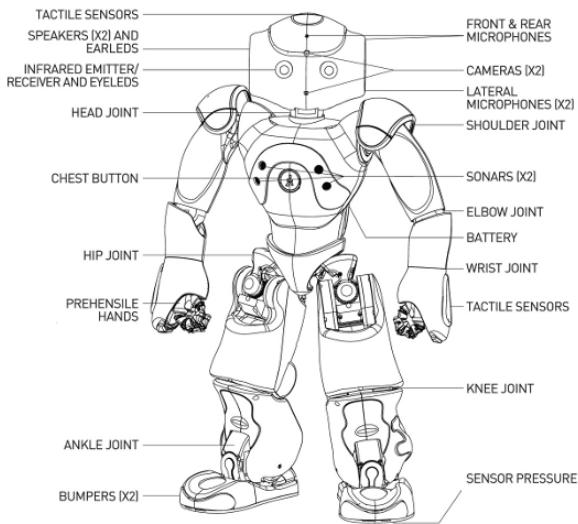


Figura 1.2: Componentes del robot NAO

El marco de trabajo de NAOqi es el marco de trabajo de programación usado para programar robots de Aldebaran (*NAO no es el único*). Brinda soluciones a necesidades básicas de la robótica como el paralelismo, el manejo de recursos, la sincronización y los eventos.

Características principales

El marco de trabajo permite la comunicación homogénea entre los diferentes módulos (movimiento, audio, video). Además es multiplataforma y multilenguaje lo que permite crear aplicaciones distribuidas.

- **Multiplataforma** : El marco de trabajo de NAOqi puede ser usado en Windows, Linux y MacOS.
- **Multilenguaje** : Con el marco de trabajo se puede desarrollar código para ejecutarlo directamente sobre el robot o remotamente en otro dispositivo. En la Tabla 1.1.2 se muestran los lenguajes compatibles.
- **Aplicaciones distribuidas** : Una aplicación puede estar compuesta de procesos y módulos entre diversos robots. Para ejecutar un programa en un robot de manera remota sólo se necesita su dirección IP y puerto, los métodos de la API son los mismos como si se llamaran localmente.

Tabla 1.1: Lenguajes de programación soportados

Lenguaje de programación	Se ejecuta por defecto sobre el robot	Se ejecuta sobre otro dispositivo (requiere el SDK)
Python	Sí	Sí
C++	Sí	Sí
Java	No	Sí
JavaScript	Sí	Sí

Introspección

La introspección es la base de la API del robot, las capacidades, el monitoreo y la acción en las funciones supervisadas. El robot conoce todas las funciones de la API disponibles. Una función definida en un módulo puede añadirse a la API con `BIND_METHOD` (definido en `almodule.h`)

El realizar la acción anterior automáticamente permite los siguiente:

- Llamar la función ya sea en C++ o en Python.
- Saber si la función está en ejecución.
- Ejecutar la función localmente o remotamente.
- Llamar a las funciones `wait`, `stop` e `isRunning`.

El proceso NAOqi

El ejecutable NAOqi en el robot es un broker. Cuando inicia, carga un archivo de preferencias llamado `autoload.ini` que se encarga de definir que bibliotecas se deberán cargar. Cada biblioteca contiene uno o más módulos que el broker usa para publicar sus métodos.

El broker provee un servicio de búsqueda para que cualquier módulo en el árbol de la Figura 1.3 o a través de una red pueda encontrar cualquier método que ha sido publicado.

La carga de módulos forma un árbol (Figura 1.4) de métodos ligados a los módulos, y módulos ligados a un broker.

Un *broker* es un objeto que provee:

- Un directorio de servicios. Permite encontrar módulos y servicios.

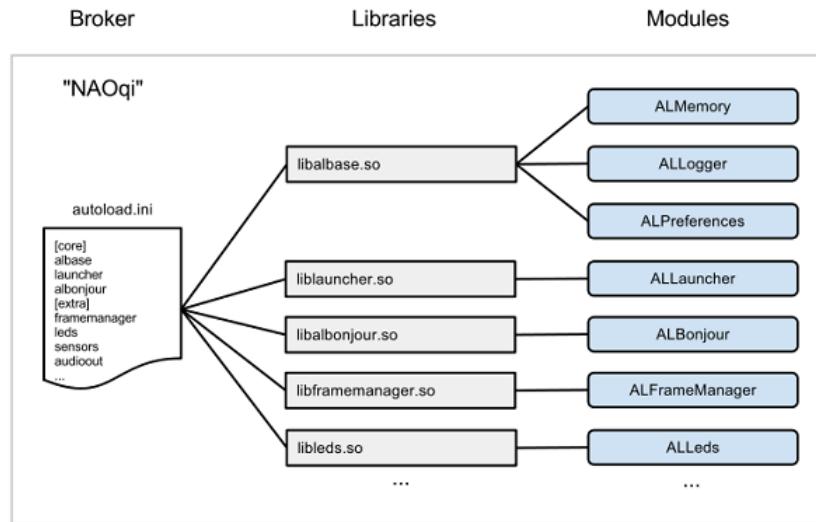


Figura 1.3: Conexión entre el broker las bibliotecas y los módulos

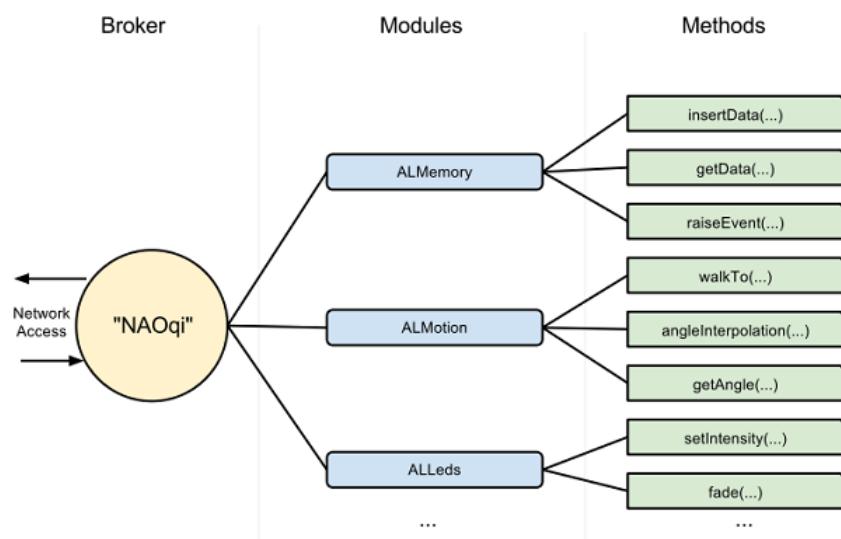


Figura 1.4: Conexión entre broker, módulos y métodos

- Acceso a la red. Permite que los métodos de módulos sean llamados desde fuera del proceso.

Un *proxy* es un objeto que se comportará como el módulo que representa. Por ejemplo si se crea un proxy al módulo **ALMotion**, se obtendrá un objeto que contiene todos los métodos de **ALMotion**.

Para crear un proxy a un módulo, se tienen dos opciones:

- Usar el nombre del módulo. El código que se ejecutará y el módulo al que se quiere conectar deben estar en el mismo broker. A esto se le conoce como llamada *local*.
- Usar el nombre del módulo, y la dirección IP y puerto de un broker. En este caso el módulo debe estar en el broker correspondiente.

Módulos

Cada módulo es una clase dentro de una biblioteca. Cuando una biblioteca se carga desde **autoload.ini**, automáticamente al crear una instancia del módulo.

En el constructor de una subclase de **ALModule**, se pueden «enlazar» o «ligar» métodos. Esto publica los nombres y firmas de sus métodos al broker para que así estén disponibles para otros.

Un módulo puede ser ya sea remoto o local. Si es remoto, es compilado como un ejecutable y puede ser ejecutado fuera del robot. Los módulos remotos son más fáciles de usar y pueden ser fácilmente depurados. Sin embargo, son menos eficientes en términos de velocidad y uso de memoria.

Si el módulo es local, se compila como una biblioteca, y puede ser usado únicamente en el robot. Sin embargo son más eficientes que los remotos.

Los *módulos locales* son dos o más módulos lanzados en el mismo proceso. Estos se comunican entre ellos usando solamente un broker.

Como los módulos locales están en el mismo proceso, estos puede compartir variables y llamar otros métodos sin serializar información y sin necesidad de una red. Esto permite la comunicación más rápida posible.

Los *módulos remotos* son aquellos que se comunican usando una red. Un módulo remoto necesita un broker para comunicarse con otros módulos. El broker es responsable de toda la parte de la red. Estos módulos trabajan usando el protocolo *SOAP* sobre una red.

Los módulos remotos puede comunicarse con otros módulos conectando su broker a los brokers de otros módulos usando un proxy.

- Una conexión **Broker a Broker** abre una comunicación mutua. Los módulos de ambos brokers pueden «hablar» entre ellos.
- Una conexión **Proxy a Broker** abre un solo canal de comunicación. El proxy puede acceder a todos los módulos registrados en el broker, pero, el módulo registrado al broker no puede acceder al módulo al que el proxy pertenece.

Se pueden conectar dos módulos conectando sus brokers. Por ejemplo, se tiene dos módulos B y C. Cuando conectas sus brokers, B puede acceder a las funciones de C y C puede acceder a las funciones de B.

Para conectar módulos de esta forma, necesitas especificar la dirección IP y el número de puerto del broker principal. Entonces se puede acceder al módulo creando un proxy de éste.

Como el broker ya fue creado con la dirección IP y el puerto, no se necesitan especificar al crear el proxy.

Se puede conectar un módulo a otro sin definir la dirección IP y el puerto. Para hacer eso, es necesario crear un proxy dentro del módulo y conectarlo a la dirección IP y puerto del broker que se desea.

Por ejemplo, si se tienen dos módulos B y C. Cuando se conecten B a C sólo usando un proxy, B puede acceder a las funciones de C pero C no puede acceder a las funciones de B.

Llamadas blocking y non-blocking

NAOqi ofrece dos maneras de llamar métodos:

- Llamadas de bloqueo (blocking call): Al llamar una función de bloqueo, la siguiente instrucción se ejecutará después de que la que fue previamente llamada. Todas las llamadas pueden lanzar excepciones y deben ser encapsuladas en bloques try-catch.
- Llamadas sin bloqueo (non-blocking call): Usando el objeto `post` de un proxy, una tarea es creada en un hilo paralelo. Esto permite hacer otro trabajo al mismo tiempo (por ejemplo, caminar y hablar). Cada `post` genera un identificador de la tarea. Se puede usar el identificador anterior para verificar si la tarea se está ejecutando, o esperar a que termine.

Memoria

`ALMemory` es la memoria del robot. Todos los módulos pueden leer o escribir datos,

suscribirse a eventos de tal forma que sean llamados cuando el evento sea lanzado. **ALMemory** no es una herramienta de sincronización en tiempo real.

ALMemory.

Es un arreglo de **ALValues**. El acceso a la variable es de hilo seguro. Se usan secciones críticas de lectura/escritura para evitar un mal desempeño cuando se lee la memoria.

ALMemory contiene tres tipos de datos:

- Datos de actuadores y sensores
- Eventos
- Micro eventos

Reacción a eventos

Algunos módulos exponen ciertos eventos. Se debe suscribir a un evento desde otro módulo, usando un callback que debe ser un método de tu suscriptor.

Por ejemplo, supongamos que se define un módulo llamado **FaceReaction** que contiene un método **onFaceDetected**. Se puede suscribir **FaceReaction** al método **FaceDetected** del módulo **ALFaceRecognition** con el callback **onFaceDetected**.

Esto provocará que el algoritmo de detección de rostros se ejecute, y cada vez que una cara sea detectada, se llame al callback **onFaceDetected**.

Interfaces de programación de aplicaciones (API) de NAOqi

NAOqi cuenta con bastantes API divididas en grupos de acuerdo a las funcionalidades que ofrecen. A continuación se enlistan estas categorías, sin embargo, sólo se describen de manera breve aquellas API, que se utilizaron en el desarrollo de este proyecto.

NAOqi viene con una lista de módulos centrales que están disponibles siempre. Cada módulo cuenta con una lista de métodos por defecto.

ALMemory. Es una memoria centralizada usada para almacenar toda la información importante relacionada con la configuración del hardware del robot.

ALMemory provee información acerca del estado actual de actuadores y sensores.

De manera más específica, **ALMemory** es un mapa no ordenado de la biblioteca **boost**. El mapa está compuesto por contenedores genéricos (**ALValues**).

La escritura y lectura es de exclusión mútua o mutex, lo que protege:

- El mapa
- Un valor
- Historial de valores

Lo que esto nos dice es que:

- Eliminar datos bloquea a todas las escrituras y lecturas.
- Actualizar un dato sólo bloquea a la información modificada.
- La lectura de datos sólo bloquea a las escrituras de los datos que se leen.

Las notificaciones son manejadas por una threadpool (localmente), o por un hilo único de notificación (remotamente).

Para acceder a los valores almacenados en **ALMemory**, su usan las funciones:

- **getDataPtr()**, que provee un acceso rápido al puntero. Sin embargo, no hay seguridad en el hilo.
- **getData()**, existe seguridad en el hilo. Puede ser usado ya sea por un módulo local o remoto.

Para guardar valores en la memoria, se utiliza **insertData()**, y **subscribeToEvent()** y **subscribeToMicroEvent()** para suscribirse a un evento y a un micro evento, respectivamente. Los eventos y micro eventos son básicamente lo mismo, pero el primero almacena un historial en **ALMemory**, el segundo por consecuencia es más rápido.

NAOqi contiene un grupo de módulos enfocados a los movimientos del robot, ya sea para navegar de manera segura, cambiar entre posturas predefinidas, realizar movimientos de forma autónoma y hasta generar movimientos personalizados.

ALMotion Es la principal herramienta para permitir que el robot se mueva.

Contiene cuatro principales grupos de métodos para controlar:

- la rigidez de articulaciones, básicamente si un motor está prendido o apagado.
- la posición de articulaciones, para la planeación de trayectorias (interpolación) y cambios en los valores de los motores como respuesta a datos en sensores (control reactivo).

- el caminado, control de distancia y velocidad, posición en un ambiente, etc.
- efecto del robot en el espacio cartesiano, determinar el movimiento de una cadena de articulaciones para lograr que un actuador se ubique en una posición concreta (cinemática inversa).

El eje **X** es positivo con respecto al frente del robot, el eje **Y** de derecha a izquierda y el **Z** es vertical. La Figura 1.5 muestra la definición de los ejes con respecto al robot. El módulo de **ALMotion** usa el Sistema Internacional de Unidades (metros, segundos, radianes, etc).

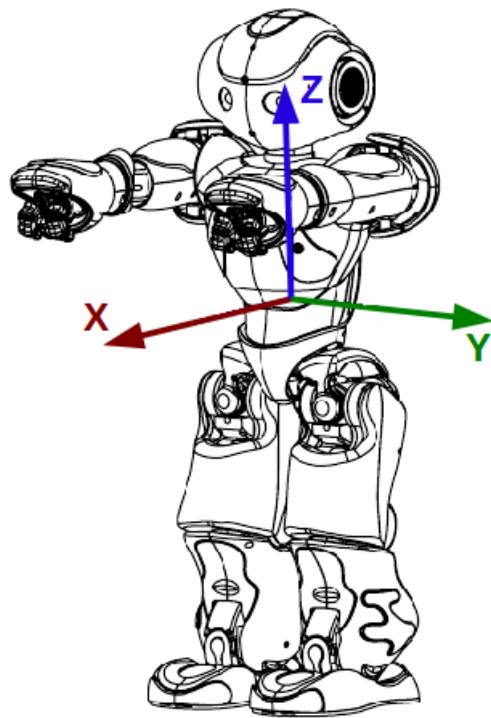


Figura 1.5: Sistema de ejes sobre los que el robot ejecuta movimientos

NAOqi cuenta con componentes de software para el audio; para manejar la salida o entrada a través de sus bocinas y micrófonos, para la detección y localización de sonidos, y para el manejo del lenguaje.

ALTextToSpeech Este módulo permite al robot hablar. Envía órdenes a un componente que convierte texto a un discurso hablado, y autoriza la personalización de la voz. El resultado de la síntesis es enviado a las bocinas del robot.

ALAnimatedSpeech El módulo ALAnimatedSpeech brinda la posibilidad de hacer que el robot hable de una manera expresiva.

El funcionamiento de este módulo es como sigue:

1. ALAnimatedSpeech recibe texto que puede ser **anotado** con instrucciones.
2. Divide el texto en subcadenas de menor tamaño a la original.
3. Analiza el texto y anota las cosas que reconoce para realizar movimientos de acuerdo al contexto.
4. Cualquier parte del texto que no esté anotado con animaciones se llena con **modos de lenguaje corporal**.
5. El módulo prepara al robot para ejecutar cada instrucción para que estas sean llamadas tan pronto como se necesiten. Esto permite que el discurso y las instrucciones estén sincronizados.
6. ALAnimatedSpeech hace que el robot diga el texto y se mantenga al tanto del discurso para lanzar las instrucciones en el momento correcto.

El texto anotado es una cadena combinando texto que se dirá e **instrucciones** que manejan comportamientos.

Por ejemplo, puedes enviar:

```
"Cells interlinked ^start(animations/Stand/Gestures/Explain_1) within
˓→cells interlinked"
```

Con esto el robot dice «Cells interlinked», luego, simultáneamente ejecuta la animación **animations/Stand/Gestures/Explain_1** mientras dice «within cells interlinked». Una vez que el robot termina el discurso, detiene la ejecución de la animación. Si se desea esperar a que termine la animación basta con añadir la instrucción **^wait** y como parámetros la animación al final del texto.

```
"Cells interlinked ^start(animations/Stand/Gestures/Explain_1) within
˓→cells interlinked ^wait(animations/Stand/Gestures/Explain_1)"
```

En cuanto a los modos del lenguaje corporal existen tres:

- **disabled**, el lenguaje corporal nunca lanza una animación.
- **random**, el lenguaje corporal ejecuta aleatoriamente algunas animaciones que son ejecutadas una tras otra.
- **contextual**, el lenguaje corporal lanza algunas animaciones específicas cada vez que una palabra clave como «Yo», «tú», es detectada en un contexto gramatical apropiado.

ALAudiorecorder `ALAudiorecorder` provee servicios de grabación en archivos «WAV» u «OGG» a partir de las señales recibidas de los micrófonos del robot.

Este módulo depende la biblioteca de Linux SDNFile para codificar de manera eficiente entradas de audio en tiempo real.

Las capacidades de grabación están limitadas a los siguientes formatos:

- cuatro canales 48000 Hz para OGG y WAV
- un canal (anterior, posterior, izquierda o derecha) para OGG y WAV.

Los módulos de visión provistos por NAOqi se encargan del manejo de video e imágenes, detección de objetos predefinidos en video, y cuenta con herramientas para crear una memoria visual donde aprenda y reconozca ciertos patrones, y herramientas para navegación, usar una imagen como brújula, entre otras.

ALVideoDevice Este módulo es responsable de proveer, de una manera eficiente, imágenes de la cámara del robot a todos los módulos que las procesan, como `ALFaceDetection` o `ALVisionRecognition`.

Para empezar a utilizar este módulo es necesario seguir los siguientes pasos:

1. Hacer que tu módulo de visión se suscriba al proxy de `ALVideoDevice`, llamando el método `subscribeCamera()` y pasando como parámetros la resolución, espacio de color y tasa de fotogramas.
2. En el bucle del proceso principal, obtener una imagen llamando a los métodos `getImageLocal()` o `getImageRemote()` (dependiendo si el módulo es local o remoto).
3. Liberar la imagen llamando `releaseImage()`.
4. Cuando se detiene el módulo, se llama `unsubscribe()` después de salir de bucle principal.

Existen numerosos sensores en el robot NAO; NAOqi ofrece módulos para interactuar con el robot a través de algunos de ellos, o manejar eventos cuando estos cambian valores.

ALSonar El módulo de `ALSonar` obtiene valores de los sensores ultrasónicos desde `ALMemory`, procesa esa información y lanza eventos de acuerdo con la situación. Para ahorrar energía, los sensores ultrasónicos no están activados por defecto.

Hay cuatro eventos diferentes relacionados con los sensores ultrasónicos, y se muestran en la tabla Tabla 1.1.2

Tabla 1.2: Eventos lanzados por los sensores ultrasónicos

Ca- so	Evento	Descripción
	SonarLeft-Detected	Hay algo enfrente del robot (lado izquierdo) al menos a 0.5 m. Esto significa que el robot no puede ir hacia adelante, tiene que detenerse y girar a la derecha para evitar el obstáculo.
	Sona- rRightDe- tected	Hay algo frente al robot (lado derecho) al menos a 0.5 m. Esto significa que el robot no puede ir hacia a delante, tiene que detenerse e ir hacia la izquierda.
	SonarLeft- Nothing- Detected	No hay presencia de un objeto en frente del robot o de su lado izquierdo, esto significa que el robot puede ir hacia adelante o girar a la izquierda. Hay un obstáculo presente a menos 0.5 m del lado derecho., pero no es problema si el robot sigue de manera recta.
	Sona- rRight- Nothing- Detected	No hay algo frente al robot o de su lado derecho. El robot puede ir hacia adelante o girar a la derecha. Hay un obstáculo a la izquierda, sin embargo no es un problema para una caminada recta.
	SonarLeft- Nothing- Detected y Sonar- rRight- Nothing- Detected	Si ambos eventos son lanzados al mismo tiempo significa que no hay obstáculos presentes.

Los valores actuales de los sensores se pueden recuperar desde `ALMemory` a través del método `getData()` de `ALMemory`.

Uso El hardware de los sensores no se inicia de manera automática. Para empezar, es necesario suscribirse al módulo de **ALSonar** a través del método `subscribe()`. Hacer esto prende los sensores ultrasónicos.

Para apagar los sensores, se debe cancelar la suscripción con el método `unsubscribe()` de **ALSonar**. Si existen múltiples suscriptores, los sensores seguirán activos tanto como el último suscriptor.

Con la función `getData()` de **ALMemory** se obtienen los valores de los sensores pasando como parámetros en la función las llaves `Device/SubDeviceList/US/Left/Sensor/Value` y `Device/SubDeviceList/US/Right/Sensor/Value` para recuperar la distancia en metros de obstáculos que detecta el sensor izquierdo y derecho respectivamente.

Rastreadores

Existe un rastreador genérico implementado en el módulo **ALTracker**, el cual permite al robot seguir diferentes objetivos predeterminados (una pelota roja, puntos de referencia, caras, etc) usando diferentes medios (la cabeza, el cuerpo completo, etc).

El principal objetivo de este módulo es establecer una relación entre la detección de un blanco y movimientos para hacer que el robot mantenga su vista sobre el objeto en el punto medio de la cámara.

Diagnóstico

NAOqi cuenta con el módulo **ALDiagnosis** que permite detectar si hay problemas en el hardware del robot, principalmente fallas en las conexiones eléctricas.

DCM

El administrador de comunicación de dispositivos, DCM por sus siglas en inglés, es un módulo de software, parte de NAOqi, encargado de la comunicación con casi todos los dispositivos electrónicos del robot (sensores, actuadores, etc), excepto los de audio y las cámaras.

El marco de trabajo qi

Es una nueva arquitectura, que permite utilizar la API de NAOqi con una sintaxis nueva y sencilla.

En vez de crear un proxy a un módulo, se crea una sesión y se solicita un servicio. El resultado de esto es un objeto con los mismos métodos del módulo que se desea.

SDK de Python

La API de Python para los robots de Aldebaran permite:

- usar la API de C++ desde una máquina remota
- crear módulos de Python que puedan ejecutarse remotamente o localmente en el robot.

Un programa muy básico en Python para los robots tiene la siguiente estructura:

- Primero importar `ALProxy`
- Instanciar un objeto de `ALProxy` para el módulo que se quiere usar
- Llamar un método

El siguiente fragmento de código es un ejemplo de un programa que hace al robot caminar sobre su eje X durante tres segundos:

```
# encoding: utf-8
import time
from naoqi import ALProxy # Importa ALProxy

motion_proxy = ALProxy("ALMotion", "<IP del robot>", 9559) # Crear un proxy al módulo ALMotion

motion_proxy.wakeUp() # Utiliza un método de ALMotion para cambiar la posición del robot
x = 0.5
y = 0.0
theta = 0.0

motion_proxy.moveToward(x, y, theta)
time.sleep(3)

motion_proxy.stopMove()

motion_proxy.rest()
```

SDK de JAVA

El SDK de Java provee una API para llamar servicios remotos, crear nuevos servicios y reaccionar a eventos.

El SDK está basado en la biblioteca **libqi-java**, la cual utiliza el **marco de trabajo qi**.

Tabla 1.3: Sistemas operativos compatibles con el SDK de Java

Sistema operativo	Notas
Windows 32 bits	No hay soporte para la versión de 64 bits
Linux 32 y 64 bit	Probado en Ubuntu 12.04 LTS
Mac OS X 64 bits	
NAOqi OS 2.1 +	Ejecutándose en las versiones de NAO V4 y V5
Android 4.0	Para ARM

Existe un archivo `.jar` por cada plataforma y versión de NAOqi. Este archivo `.jar` permite usar la API de C++ de Aldebaran o crear servicios personalizados, y ejecutarlos desde un dispositivo remoto o directamente en el robot.

Para compilar una aplicación desde la línea de comandos, se realiza los siguiente

```
# Compilación  
javac -cp /ruta/a/java-naoqi-sdk-<version>-<platform>.jar MyApp.java  
# Ejecución  
java -cp /ruta/a/java-naoqi-sdk-<version>-<platform>.jar:. MyApp
```

Hay dos conceptos importantes que se deben conocer que permiten la comunicación con el robot: **Application** y **Session**.

Una **Application** es responsable de inicializar el marco de trabajo y conectarse a una sesión. Una **Session** es lo que permite conectarse a servicios local o remotamente.

Por defecto, el URL de la sesión está configurado para ser `tcp://127.0.0.1:9559`. Pero éste se puede cambiar en los argumentos del constructor de **Application**.

En el SDK de Java, existen clases para cada servicio. Esto último significa que se debe crear una instancia de cada servicio que se desea para que sea posible llamar sus métodos.

El siguiente ejemplo muestra como hacer que el robot diga la frase “Cells Interlinked Within Cells Interlinked” usando **ALTextToSpeech**.

```
public static void main(String[] args) {  
    // Se define el URL del robot
```

```

String robotUrl = "tcp://192.168.4.220:9559";
Application application = new Application(args, robotUrl);
try{
    // Se inicial la aplicación y se genera una sesión
    application.start();
    // La sesión puede recuperarse a través del metodo siguiente
    // application.session();
    // Crear un objeto de ALTextToSpeech y lo enlaza con la sesión
    ↵actual
    ALTextToSpeech tts = new ALTextToSpeech(application.session());
    // El robot dice la frase
    tts.say("Cells Interlinked Within Cells Interlinked");
}
catch(Exception e){
    // La aplicación no pudo iniciar.
    e.printStackTrace();
}
}

```

1.2. Algoritmos de aprendizaje profundo y TensorFlow

1.2.1. Aprendizaje automático

Remark. Un algoritmo de aprendizaje automático es un algoritmo que tiene la capacidad de aprender a partir de datos.

Remark. Un programa de computadora se dice que aprende de la experiencia E con respecto a alguna clase de tarea T y una medida de desempeño P , si su desempeño de tareas en T , medido por P , mejora con la experiencia E .

La tarea T

Las tareas son difíciles de resolver con programas estáticos escritos y diseñados por humanos. Aprender es el medio para tener la habilidad de realizar una tarea.

Las tareas del aprendizaje automático son descritas en términos de cómo el sistema de aprendizaje automático debe procesar una muestra.

Remark. Una muestra es una colección de características que han sido medidas cuantitativamente a partir de un objeto o evento que queremos que nuestro sistema de aprendizaje automático procese. De manera típica una muestra se representa como un vector $\mathbf{x} \in \mathbb{R}^n$ donde cada elemento x_i del vector es una característica.

Muchos tipos de tareas se pueden resolver con aprendizaje automático. Algunas de las tareas más comunes en el aprendizaje automático incluyen las siguientes:

- **Clasificación:** En este tipo de tarea, se le pregunta al programa a cuál de k categorías pertenece una entrada. Para resolver esta tarea, el algoritmo de aprendizaje generalmente produce una función $f : \mathbb{R}^n \rightarrow \{1, \dots, k\}$. Cuando $y = f(\mathbf{x})$, el modelo asigna una entrada descrita por el vector \mathbf{x} a una categoría identificada por un código numérico y .
- **Clasificación con entradas faltantes:** La clasificación se vuelve más difícil si al programa no se le garantiza que cada medida en su vector de entradas, será dada siempre. Para resolver la tarea de clasificación, el algoritmo de aprendizaje sólo tiene que definir una función que mapee de un vector de entradas a una salida categórica. Cuando algunas entradas faltan, más que proveer una sola función de clasificación, el algoritmo de aprendizaje debe aprender de un conjunto de funciones. Cada función corresponde a clasificar \mathbf{x} con un subconjunto diferente de entradas faltantes. Una manera eficiente de definir un conjunto tan grande de funciones es aprender la distribución de probabilidad sobre todas las variables relevantes, entonces resolver la tarea de clasificación marginalizando las variables faltantes. Con n variables de entrada, podemos obtener todas las 2^n funciones de clasificación que se necesitan para cada conjunto de entradas faltantes posible, pero sólo se necesita aprender una función que describa la distribución de probabilidad conjunta.
- **Regresión:** En este tipo de tarea, al programa se le solicita predecir un valor numérico dada una entrada. Para resolver esta tarea, al algoritmo de aprendizaje se le pide que genere una función $f : \mathbb{R}^n \rightarrow \mathbb{R}$. Este tipo de tarea es similar al de clasificación, excepto que el formato de la salida es diferente.
- **Transcripción:** En este tipo de tarea, al sistema de aprendizaje automático se le pide observar la representación relativamente no estructurada de algún tipo de información y transcribirlo de una forma textual y discreta.
- **Traducción máquina:** En una tarea de traducción máquina, la entrada consiste de una secuencia de símbolos en algún lenguaje, y el programa de computadora debe convertir esto en una secuencia de símbolos en otro lenguaje.
- **Salida estructurada:** Las tareas de salida estructurada involucran cualquier tarea donde la salida es un vector (o cualquier otro tipo de estructura de datos que contenga múltiples valores) con relaciones importantes entre los

diferentes elementos. Esta categoría subsume las tareas como la traducción y transcripción. Estas tareas son llamadas así porque el programa debe producir varios valores que están estrechamente interrelacionados.

- **Detección de anomalías:** En este tipo de tarea, el programa examina cuidadosamente un conjunto de eventos u objetos, y marca algunos de ellos por ser atípicos o inusuales.
- **Síntesis y muestreo (sampling):** En este tipo de tarea se le solicita al algoritmo de aprendizaje automático que genere nuevas muestras que son similares a las de un conjunto de entrenamiento. La síntesis y muestreo a través de aprendizaje automático, puede ser útil en aplicaciones multimedia donde puede ser caro o cansado para un artista generar grandes volúmenes de contenido a mano. En algunos casos, se quiere que el procedimiento de muestreo y síntesis genere algún tipo de salida específica dada la entrada.
- **Atribución de valores faltantes:** En este tipo de tarea, al algoritmo de aprendizaje máquina se le da una nueva muestra $\mathbf{x} \in \mathbb{R}^n$, pero con algunos elementos x_i de \mathbf{x} faltantes. El algoritmo debe predecir los valores de las entradas faltantes.
- **Reducción de ruido:** En este tipo de tarea, el algoritmo de aprendizaje automático recibe como una entrada una muestra corrupta $\mathbf{x}' \in \mathbb{R}^n$, obtenido por un proceso de corrupción desconocido de una muestra limpia $\mathbf{x} \in \mathbb{R}^n$. El aprendiz debe predecir la muestra limpia \mathbf{x} de su versión corrupta \mathbf{x}' , o de manera más general predecir la distribución de probabilidad condicional $P(\mathbf{x}|\mathbf{x}')$.

La medida de desempeño P

A fin de que se puedan evaluar las habilidades de un algoritmo de aprendizaje automático, se debe diseñar una medida cuantitativa de su desempeño. Usualmente esta medida de desempeño P es específica a la tarea T que se lleva a cabo por el sistema.

Para tareas de clasificación, clasificación con entradas faltantes, y transcripción, se mide la precisión (*accuracy*) del modelo.

Remark. *La precisión es sólo la proporción de muestras para los cuales el modelo produce la salida correcta. Se puede obtener una información equivalente midiendo la tasa de error, la proporción de muestras para las cuales el modelo produce la salida incorrecta.*

Usualmente estamos interesados cómo se desempeña un algoritmo de aprendizaje

automático sobre datos que no ha visto, ya que determina que tan bien trabajará cuando se lance al mundo real. Por lo tanto evaluamos estas medidas de desempeño usando un *conjunto de prueba*, que está separado de los datos usados para entrenar el sistema de aprendizaje automático.

La experiencia E

Los algoritmos de aprendizaje automático pueden ser de manera muy general categorizados como *supervisados* o *no supervisados* por el tipo de experiencia que tienen disponible durante el proceso de aprendizaje.

A la mayoría de los algoritmos de aprendizaje se les permite experimentar un *conjunto de datos*.

Remark. *Un conjunto de datos es una colección de muchas muestras. Algunas veces a esas muestras se les llama patrones de entrada. De manera muy general un conjunto de datos es una colección de muestras, que a su vez son colecciones de características.*

Remark. *Los algoritmos de aprendizaje no supervisados experimentan un conjunto de datos que contiene una gran cantidad de características, entonces aprenden algunas propiedades útiles de la estructura del conjunto de datos.*

Algunos algoritmos no supervisados realizan papeles, como el de agrupamiento, que consiste en dividir el conjunto de datos en grupos de muestras similares.

Remark. *El aprendizaje no supervisado involucra observar varias muestras de un vector aleatorio \mathbf{x} , e intentar aprender implícitamente o explícitamente la distribución de probabilidad $P(\mathbf{x})$, de algunas propiedades interesantes de la distribución.*

Algoritmos de aprendizaje supervisados

Remark. *Los algoritmos de aprendizaje supervisados experimentan un conjunto de datos que contiene características, pero cada muestra está asociada además con una etiqueta u objetivo. El aprendizaje supervisado involucra observar bastantes muestras de un vector aleatorio \mathbf{x} y un valor o vector asociado \mathbf{y} , y aprender a predecir \mathbf{y} a partir de \mathbf{x} , usualmente estimando $P(\mathbf{y}|\mathbf{x})$.*

Capacidad, sobreajuste e infra-ajuste

El principal reto del aprendizaje automático es que se debe desempeñar bien sobre entradas nuevas no vistas previamente, no sólo con las que el modelo fue entrenado.

La habilidad de desempeñarse bien sobre entradas no observadas previamente se llama *generalización*.

Comúnmente cuando se entrena un modelo de aprendizaje automático, se tiene acceso a un *conjunto de entrenamiento*, se puede calcular alguna medida de error sobre el conjunto de entrenamiento llamado error de entrenamiento, y reducir este error. Hasta acá, simplemente se ha descrito un problema de optimización ya que se pretende encontrar la mejor solución. Lo que distingue al aprendizaje automático de la optimización es que queremos que el *error de generalización*, también llamado error de prueba, sea bajo también.

Remark. *El error de generalización se define como el valor esperado del error de una nueva entrada. Aquí la esperanza es tomada a través de diferentes entradas posibles. extraída de la distribución de entradas que esperamos el sistema se encuentre en la práctica.*

Típicamente estimamos el error de generalización de un modelo de aprendizaje automático midiendo su desempeño sobre un conjunto de prueba de muestras que fueron recolectadas de manera separada del conjunto de entrenamiento.

¿Cómo se puede afectar el desempeño sobre el conjunto de pruebas cuando solo se ha observado el conjunto de entrenamiento? Mediante un *proceso de generación de datos* los conjuntos de entrenamiento y de prueba son producidos por una distribución de probabilidad sobre conjuntos de datos. Se hacen un conjunto de suposiciones conocidas colectivamente como las suposiciones i.i.d. Estas suposiciones son que las muestras en cada conjunto de datos son independientes entre sí, y los conjuntos de entrenamiento y de prueba están idénticamente distribuidos. Esta suposición nos permite describir el proceso de generación de datos con una distribución de probabilidad sobre una sola muestra. La misma distribución es usada para generar cada muestra de entrenamiento y cada muestra de prueba. Llamamos a la distribución mencionada la distribución de generación de datos, denotada por p_{data} . Este marco de trabajo probabilístico y las suposiciones i.i.d. permiten estudiar matemáticamente la relación entre el error de entrenamiento y el error de prueba.

Se puede observar que el valor esperado del error de entrenamiento de un modelo seleccionado aleatoriamente es igual al valor esperado del error de prueba del modelo. Supongamos que tenemos una distribución de probabilidad $P(x, y)$ y obtenemos muestras de esta repetidamente para generar el conjunto de entrenamiento y el conjunto de prueba. Para algún valor fijo w , el valor esperado del error de entrenamiento es exactamente el mismo que el valor esperado del conjunto de prueba, porque ambas esperanzas están formadas usando el mismo proceso de muestreo. La única diferencia entre las dos condiciones es el nombre que asignamos al conjunto de datos que muestreamos.

Cuando usamos un algoritmo de aprendizaje automático, no se fijan los parámetros antes de tiempo y luego se hace el muestreo de los conjuntos de datos. Muestramos el conjunto de entrenamiento, entonces se eligen los parámetros para reducir el error de entrenamiento, luego se hace el muestreo del conjunto de prueba. Los factores que determinan que tan bien se desempeña un algoritmo de aprendizaje automático son la habilidad de:

- Minimizar el error de entrenamiento.
- Minimizar la diferencia entre el error de entrenamiento y el de prueba.

Estos dos factores corresponden a los dos retos principales en el aprendizaje automático: *sobreajuste* e *infra-ajuste*.

Remark. *El infra-ajuste ocurre cuando el modelo no puede obtener un valor en el error lo suficientemente bajo sobre el conjunto de entrenamiento. El sobreajuste ocurre cuando la diferencia entre el error de entrenamiento y de prueba es muy grande.*

Se puede controlar cuando un modelo es más propenso a sobre ajustarse o infra-ajustarse alterando su capacidad.

Remark. *La capacidad de un modelo es la habilidad de ajustarse a una amplia variedad de funciones. Modelos con baja capacidad pueden esforzarse por ajustarse a su conjunto de entrenamiento. Los modelos con alta capacidad pueden sobre ajustarse memorizando propiedades del conjunto de entrenamiento que no pueden funcionar bien sobre el conjunto de prueba.*

Una manera de controlar la capacidad de un algoritmo de aprendizaje es eligiendo un conjunto de hipótesis, el conjunto de funciones de donde el algoritmo de aprendizaje puede seleccionar la solución. Los algoritmos de aprendizaje automático generalmente se desempeñarán mejor cuando su capacidad es apropiada en lo que respecta a la verdadera complejidad de la tarea que necesitan realizar y la cantidad de datos de entrenamiento que utilizan. Modelos con capacidad insuficiente son incapaces de resolver tareas complejas. Modelos con alta capacidad pueden resolver tareas complejas, pero cuando su capacidad es mayor de la necesaria para resolver la tarea, tienden a sobre ajustarse.

La capacidad no está determinada únicamente por la elección del modelo. El modelo especifica qué familia de funciones el algoritmo de aprendizaje puede elegir donde se varían los parámetros para reducir el objetivo de entrenamiento. Esa es llamada la *capacidad representacional* del modelo. En muchos casos, la mejor función dentro de esa familia de funciones es un problema de optimización muy difícil. En la práctica, el algoritmo de aprendizaje no encuentra la mejor función, pero sí una que reduce el error de entrenamiento significativamente. Estas limitantes adicionales como la

imperfección del algoritmo de optimización, significan que la capacidad efectiva puede ser menor que la capacidad representacional de familia del modelo.

Optimización basada en el gradiente

La mayoría de los algoritmos de aprendizaje involucran la optimización de alguna forma. La optimización se refiere a la tarea de ya sea minimizar o maximizar una función $f(\mathbf{x})$ alterando \mathbf{x} .

La función que queremos minimizar o maximizar se conoce como *función objetivo*. En el contexto del aprendizaje automático, queremos minimizar una *función de costo* que mide el error de entrenamiento.

Antes de avanzar conviene repasar algunos conceptos de cálculo relacionados con la optimización. Supongamos que tenemos la función $y = f(x)$, donde x y y son números reales. La *derivada* de la función se denota como $f'(x)$ o $\frac{dy}{dx}$. La derivada $f'(x)$ da la pendiente de la recta tangente de $f(x)$ en el punto x . La derivada especifica como se puede escalar un cambio menor en la entrada de manera que se obtenga el cambio correspondiente en la salida: $f(x + \gamma) \approx f(x) + \gamma f'(x)$.

La derivada por lo tanto es útil para minimizar una función porque nos dice cómo cambiar x para hacer un decremento en y . Por ejemplo, sabemos que $f(x - \gamma \operatorname{sgn}(f'(x)))$ es menor que $f(x)$ para un γ lo suficientemente menor. Podemos reducir $f(x)$ moviendo x en pequeños pasos con el signo opuesto a la derivada. Esta técnica se llama *descenso del gradiente*.

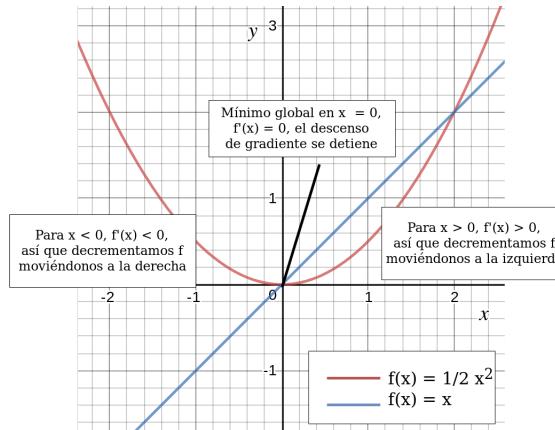


Figura 1.6: El método del descenso del gradiente utiliza las derivadas de una función para ir cuesta abajo hasta llegar a un mínimo.

Cuando $f'(x) = 0$ la derivada no provee ninguna información sobre la dirección en la cual moverse. Puntos donde $f'(x) = 0$ se conocen como *puntos críticos*. Un

mínimo local es un punto donde $f(x)$ es menor que todos sus puntos vecinos, así que no es posible disminuir $f(x)$ haciendo pasos infinitesimales. Un *máximo local* es un punto donde $f(x)$ es mayor que todos los puntos vecinos, así que no es posible incrementar $f(x)$ con pasos infinitesimales. Algunos puntos críticos no son ni máximos ni mínimos, son llamados *puntos de silla*.

Un punto que obtiene el valor menor absoluto de $f(x)$ es un *mínimo global*. Es posible que un mínimo local no sea globalmente óptimo. En el contexto del aprendizaje profundo, se optimizan funciones que pueden tener mínimos locales que no son óptimos, y puntos de silla atrapados por regiones planas. Esto hace el proceso de optimización muy complicado, especialmente si la función es multidimensional. Por esto usualmente se busca un valor de f muy bajo, pero no necesariamente el mínimo en un sentido formal.

Comúnmente se minimizan funciones que tienen múltiples entradas $f : \mathbb{R}^n \rightarrow \mathbb{R}$. Para funciones con varias entradas, se hace uso del concepto de *derivada parcial*. La derivada parcial $\frac{\partial}{\partial x_i} f(\mathbf{x})$ mide cómo cambia f cuando sólo la variable x_i se incrementa en el punto \mathbf{x} .

Remark. *El gradiente generaliza la noción de derivada para el caso donde la derivada es con respecto a un vector: el gradiente de f es el vector que contiene todas las derivadas parciales, denotado por $\nabla f(\mathbf{x})$. El i -ésimo elemento del gradiente es la derivada parcial de f con respecto a x_i . En múltiples dimensiones, los puntos críticos son puntos donde cada elemento del gradiente es igual a cero.*

Remark. *La derivada direccional mide la tasa de cambio de una función en un punto sobre un vector. La derivada direccional en la dirección \mathbf{u} (un vector unitario) es la pendiente de la función f en la dirección \mathbf{u} y se denota como $D_{\mathbf{u}}f(\mathbf{x})$.*

Para minimizar f , nos gustaría encontrar la dirección en la que f disminuye más rápidamente. Esto se puede hacer a través de la derivada direccional.

$$\begin{aligned} D_{\mathbf{u}}f(\mathbf{x}) &= \nabla f(\mathbf{x}) \cdot \mathbf{u} \\ &= \|\nabla f(\mathbf{x})\| \|\mathbf{u}\| \cos(\theta) \\ &= \|\nabla f(\mathbf{x})\| \cos(\theta) \end{aligned}$$

se puede ver que el valor máximo de $D_{\mathbf{u}}f(\mathbf{x})$ se alcanza cuando $\cos(\theta) = 1$. Por lo tanto $\theta = 0$, y el valor máximo de la derivada direccional se tiene cuando \mathbf{u} tiene la misma dirección que $\|\nabla f(\mathbf{x})\|$. Este valor máximo de $D_{\mathbf{u}}f(\mathbf{x})$ es

$$\|\nabla f(\mathbf{x})\| \cos(\theta) = \|\nabla f(\mathbf{x})\|$$

De igual forma, el valor mínimo de $D_{\mathbf{u}}f(\mathbf{x})$ puede obtenerse haciendo $\theta = \pi$ de manera que \mathbf{u} apunte a la dirección opuesta de $\nabla f(\mathbf{x})$.

Remark. La derivada direccional $D_{\mathbf{u}} f(\mathbf{x})$ tiene un valor máximo $\|\nabla f(\mathbf{x})\|$ cuando \mathbf{u} tiene la misma dirección que $\nabla f(\mathbf{x})$ y un mínimo valor $-\|\nabla f(\mathbf{x})\|$ cuando \mathbf{u} tiene la dirección que $-\nabla f(\mathbf{x})$. En otras palabras, el vector gradiente apunta en la dirección de la máxima tasa de crecimiento de la función y el vector gradiente negativo apunta a la máxima tasa de disminución de la función.

Remark. Podemos disminuir f moviéndonos en la dirección del gradiente negativo. Esto es conocido como el método del descenso más empinado o descenso del gradiente.

El descenso del gradiente propone un nuevo punto

$$\mathbf{x}' = \mathbf{x} - \gamma \nabla f(\mathbf{x})$$

donde γ es la tasa de aprendizaje, un escalar positivo que determina el tamaño de los pasos. Se puede elegir γ de diferentes maneras, pero es muy común que γ se defina como una constante pequeña.

El descenso del gradiente converge cuando cada elemento del gradiente es cero (en la práctica son valores muy cercanos a cero).

Resumiendo, la manera en la que el método del descenso del gradiente trabaja, es calculando repetidamente el gradiente $\nabla f(\mathbf{x})$, y después se moviéndose en la dirección opuesta. Con esto se disminuirá f hasta posiblemente alcanzar un mínimo global.

Terminología

A continuación se definen diferentes términos que se utilizan a lo largo del documento.

- **Ejemplos o muestras:** objetos o instancias de información usadas para aprender o evaluar.
- **Características:** El conjunto de atributos, comúnmente representadas como un vector, asociado a una muestra.
- **Etiquetas:** Valores o categorías asignadas a muestras.
- **Muestra de entrenamiento:** Las muestras usadas para entrenar un algoritmo.
- **Muestra de validación:** Muestras usadas para ajustar los parámetros de un algoritmo de aprendizaje cuando se trabaja con datos etiquetados. Los algoritmos de aprendizaje usualmente tienen uno o más parámetros libres, y

la muestra de validación es usada para seleccionar los valores apropiados para los parámetros del modelo.

- **Muestra de prueba:** Muestras usadas para evaluar el desempeño de un algoritmo de aprendizaje. La muestra de prueba está separada del conjunto de entrenamiento y validación, y no está disponible en la etapa de aprendizaje.
- **Función de costo:** También llamada función de error o de pérdida. Es una función que mide la diferencia, o pérdida, entre la etiqueta predicha y la verdadera. Denotando al conjunto de todas las etiquetas como Y y al conjunto de todas las posibles predicciones como Y' , una función C es una mapeo $C : Y \times Y' \rightarrow \mathbb{R}^+$. En la mayoría de los casos $Y = Y'$ y la función de pérdida está acotada, pero estas condiciones no se cumplen siempre. Algunos ejemplos comunes de funciones de pérdida incluyen la 0–1, definida como $C(y, y') = 1$ si $y \neq y'$ y la pérdida cuadrada definida por $C(y, y') = (y' - y)^2$.
- **Conjunto de hipótesis:** Un conjunto de funciones que mapean características (vectores de características) al conjunto de etiquetas Y . De manera más general, las hipótesis pueden ser funciones que mapean las características a un conjunto diferente Y' .

1.2.2. Redes neuronales artificiales

En su forma más general, una *red neuronal artificial* o simplemente *red neuronal*, es una máquina diseñada para modelar la manera en la cual el cerebro realiza una tarea en particular o una función de interés. Las redes neuronales artificiales utilizan una enorme cantidad de interconexiones de células de cómputo conocidas como neuronas o unidades de procesamiento.

Remark. Una red neuronal artificial es un procesador paralelo distribuido construido de unidades simples de procesamiento, que tiene una naturaleza propensa a almacenar conocimiento basado en experiencias y poniéndolo a disposición para su uso. Se asemeja al cerebro en dos aspectos:

1. El conocimiento es adquirido por la red desde su ambiente a través de un proceso de aprendizaje. El proceso de aprendizaje se hace a través de un algoritmo de aprendizaje.
2. Las fuerzas de conexión entre neuronas, también conocidas como pesos sinápticos, son usadas para almacenar el conocimiento adquirido.

En términos generales, una red neuronal artificial consiste en un gran número de procesadores simples enlazados por conexiones ponderadas. Cada unidad recibe entradas que vienen de muchas otras unidades y produce un valor escalar como salida

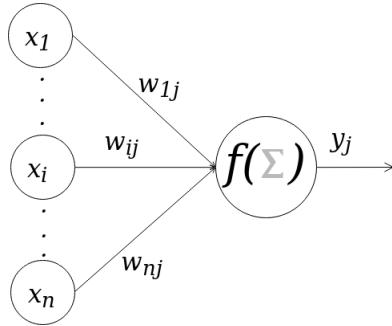


Figura 1.7: Un modelo de una neurona, la unidad más simple de procesamiento en la red.

que depende sólo de la información disponible localmente, ya sea de la que guarda internamente o la que llega a través de las conexiones ponderadas. La información es distribuida y actúa como entrada a otras unidades de procesamiento.

Por sí mismo, un elemento de procesamiento no es muy poderoso; el poder del sistema surge de la combinación de muchas unidades en una red. Una red está especializada en implementar diferentes funciones cambiando las topología de las conexiones en la red y los valores de las conexiones ponderadas.

Las unidades de procesamiento tienen respuestas como

$$y = f\left(\sum_k w_k x_k\right) \quad (1.1)$$

donde \$x_k\$ son las señales de salida de otros nodos o entradas de un sistema externo, \$w_k\$ son los pesos de los enlaces de conexión y \$f\$ es una función no lineal. Aquí una la unidad calcula una combinación lineal de los pesos y sus entradas y la pasa por \$f\$ para producir un valor escalar. \$f\$ es conocida como *función de activación*, y es muy común que se utilice una función no lineal acotada y creciente como la sigmoide.

$$f(u) = \frac{1}{1 + e^{-u}}$$

El término perceptrón a menudo se utiliza para referirse a cualquier red de nodos *feedforward* con respuestas como la ecuación (1.1). Una red puede tener cualquier estructura arbitraria, sin embargo, las arquitecturas en capas son muy populares.

Remark. *El perceptrón multicapa o MLP (por sus siglas en inglés), es ampliamente utilizado. En tales estructuras las unidades en la capa \$L\$ reciben entradas de la capa que les precede \$L-1\$ y envía sus salidas a la capa siguiente \$L+1\$. Entradas externas se presentan en la primera capa y las salidas del sistema se toman de la última capa.*

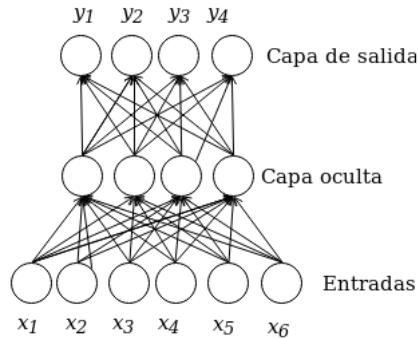


Figura 1.8: Una red neuronal feedforward completamente conectada con múltiples capas.

Las capas internas se llaman capas ocultas. Las redes más simples tienen una sola capa activa, las de las unidades de salida, por convención, las entradas no se cuentan como una capa activa ya que no realizan algún tipo de procesamiento. Las redes con una sola capa son menos poderosas que las multicapas por lo que sus aplicaciones están muy limitadas.

Que una estructura sea *feedforward* o *prealimentada* significa que no existen bucles en la red, la información siempre alimenta hacia adelante, nunca hacia atrás. La red implementa un mapeo estático que depende únicamente de las entradas que se presenten y es independiente de los estados previos del sistema.

Remark. *En una red completamente conectada, cada nodo en la capa L recibe entradas de cada nodo en la capa anterior $L - 1$ y envía su salida a todos los nodos en la capa $L + 1$.*

Las redes neuronales pueden verse como gráficas cuyos nodos son unidades de cálculo y cuyas aristas transmiten información numérica de nodo a nodo. Cada unidad de cómputo es capaz de evaluar su entrada en una función de activación. La red representa una cadena de funciones compuestas, que transforman una entrada en un vector de salida. La red es una implementación particular de una función compuesta desde la entrada hasta el espacio de salida, a la cual llamamos *función de red*.

En el contexto del aprendizaje automático supervisado, la experiencia de la que aprenden las redes neuronales es un conjunto con patrones de entrada donde cada patrón tiene una salida deseada. Nuestro problema de aprendizaje consiste en encontrar la combinación de pesos óptima tal que la función de red φ se aproxime lo más que se pueda a una función F . Sin embargo, no contamos con la función F explícitamente, sino sólo a través de algunas muestras.

Consideremos una red prealimentada con n unidades de entrada y m unidades

de salida. Ésta consiste de cualquier número de unidades ocultas y de cualquier topología en sus conexiones. También contamos con un conjunto de entrenamiento $\{(\mathbf{x}_1, \mathbf{t}_1), \dots, (\mathbf{x}_p, \mathbf{t}_p)\}$, con p pares ordenados de vectores de tamaño n y m , los cuales se llaman patrones de entrada y de salida. Definamos a las funciones de activación de cada nodo en la red como continuas y diferenciables. Los pesos de las aristas son números reales seleccionados aleatoriamente. Cuando el patrón de entrada \mathbf{x}_i del conjunto de entrenamiento se presenta en la red, produce una salida \mathbf{y}_i , generalmente diferente al objetivo \mathbf{t}_i . Queremos que \mathbf{y}_i y \mathbf{t}_i sean idénticos para $i = 1, \dots, p$, usando un algoritmo de aprendizaje. De manera más precisa, queremos minimizar una *función de costo* de la red con respecto a sus pesos; por ejemplo la definida como sigue:

$$E = \frac{1}{2} \sum_{i=1}^p \|\mathbf{t}_i - \mathbf{y}_i\|^2$$

Después de minimizar esta función para el conjunto de entrenamiento, nuevos patrones de entrada desconocidos alimentan a la red y esperamos que se interpolen. La red debe reconocer cuando un nuevo vector de entrada es similar a los patrones aprendidos y producir una salida semejante.

El *algoritmo de propagación hacia atrás o retropropagación* es el método para entrenar redes neuronales más utilizado y se describe en la siguiente sección.

Retropropagación

El término retropropagación se refiere a dos cosas diferentes. Primero, describe un método para calcular las derivadas de la función de costo con respecto a los pesos aplicando la regla de la cadena. Segundo, describe un algoritmo de entrenamiento, equivalente al descenso del gradiente; el gradiente de la función de costo es calculado y usado para corregir los pesos iniciales, que fueron elegidos aleatoriamente.

Como algoritmo de entrenamiento, el propósito de la retropropagación es ajustar los pesos de la red para producir la salida deseada como resultado a cada patrón de entrada en un conjunto de patrones de entrenamiento. Es un algoritmo *supervisado* en el sentido que, para cada patrón de entrada, existe exactamente una salida correcta.

Para entrenar una red, es necesario tener un conjunto de patrones de entrada y sus salidas deseadas correspondientes, además una función de error que mide el “costo” de las diferencias entre las salidas de la red y los valores deseados. De manera muy general, los pasos básicos del algoritmo son los siguientes:

1. Inicializar los pesos de manera aleatoria con valores pequeños.

2. Presentarle a la red una muestra del conjunto de entrenamiento para obtener las salidas. Este paso también es conocido como feedforward o propagación hacia adelante.
3. Comparar las salidas con los valores deseados y calcular el error.
4. Calcular las derivadas del error con respecto a cada uno de los pesos $\frac{\partial E}{\partial w_{ij}}$.
5. Ajustar los pesos para minimizar el error.
6. Repetir.

Propagación hacia adelante

Por simplicidad, supongamos que los nodos están indexados tal que $i < j$ implica que el nodo j sigue al nodo i en términos de dependencia. Esto quiere decir que, el estado del nodo j puede depender, quizás indirectamente, del estado del nodo i , pero el nodo i no depende del nodo j . Esta notación permite que durante las simulaciones se evite la necesidad de lidiar con cada capa de manera separada, al hacer un seguimiento de los índices de la capa. Por supuesto, este esquema de indexado es compatible con las estructuras multicapas.

En el paso hacia adelante, la red calcula una salida basada en sus entradas actuales. Cada nodo j calcula una suma ponderada a_j de sus entradas y la pasa a través de una función para obtener la salida del nodo y_j .

$$a_j = \sum_{i < j} w_{ij} y_i$$

$$y_j = f(a_j)$$

w_{ij} denota el peso que llega al nodo j desde el nodo i . El índice i en la suma va sobre todos los índices $i < j$ de nodos que envían una entrada al nodo j . Normalmente la función f es la sigmoide, sin embargo, no es la única función de activación.

Cada nodo es evaluado en orden, comenzando con el primer nodo oculto y continuando hasta llegar al último nodo de salida. En redes con múltiples capas, la primera capa oculta se actualiza basándose en las salidas de los nodos de entrada, que son los valores de un vector de características de una muestra; la segunda capa oculta se actualiza basándose en las salidas de la primera capa oculta, y se continúa así hasta llegar a la capa de salida la cual se actualiza con las salidas de la última capa oculta.

Cálculo del error

A menos que la red esté perfectamente entrenada, las salidas de la red diferirán de las salidas deseadas. Como ya vimos, para medir esa diferencia, se utiliza una función de costo, que por ahora será la *suma de cuadrados del error* o *SSE* (por sus siglas en inglés).

$$E = \frac{1}{2} \sum_p \sum_k (t_{pk} - y_{pk})^2$$

donde p indexa a todos los patrones del conjunto de entrenamiento, k indexa a los nodos de salida, t_{pk} y y_{pk} son respectivamente, el objetivo y la salida actual de la red para el k ésmo nodo de salida de la muestra p . Una de las razones por las que la SSE es conveniente es porque los errores entre las diferentes muestras o patrones y las diferentes salidas son independientes, el error total es la suma de los errores cuadrados individuales.

$$E = \sum_p E_p$$

donde

$$E_p = \frac{1}{2} \sum_k (t_{pk} - y_{pk})^2$$

Cálculo de las derivadas

Después de obtener las salidas y de haber calculado el error, el siguiente paso es calcular la derivada del error con respecto de los pesos. Recordando que $E = \sum_p E_p$ es la suma de los errores individuales de los patrones, entonces la derivada total es sola la suma de las derivadas por muestra.

$$\frac{\partial E}{\partial w_{ij}} = \sum_p \frac{\partial E_p}{\partial w_{ij}}.$$

Lo que hace eficiente a la retropropagación (el cálculo de la derivada) es cómo se descompone la operación y el orden de los pasos.

Conviene calcular de forma separada las derivadas del error con respecto a los pesos que se conectan a la unidad de salida y para las conexiones de los nodos ocultos.

La derivada con respecto a las conexiones a las unidades de salida puede ser escrita como

$$\frac{\partial E_p}{\partial w_{jk}} = \frac{\partial E_p}{\partial y_k} \frac{\partial y_k}{\partial a_k} \frac{\partial a_k}{\partial w_{jk}} \quad (1.2)$$

donde k indexa a una unidad de salida. Conviene primero calcular un valor δ_k para cada nodo de salida k . Este valor delta también es conocido como *error de retropropagación*.

$$\begin{aligned}\delta_k &= \frac{\partial E_p}{\partial y_k} \frac{\partial y_k}{\partial a_k} \\ &= -(t_k - y_k) f'(a_k).\end{aligned}$$

Para el tercer término de (1.2), como a_k es una suma lineal, es cero si $i \neq j$, de otra forma

$$\begin{aligned}\frac{\partial a_k}{\partial w_{jk}} &= \frac{\partial \sum_i w_{ik} y_i}{\partial w_{jk}} \\ &= y_j.\end{aligned}$$

Por lo tanto

$$\frac{\partial E_p}{\partial w_{jk}} = \delta_k y_j$$

El segundo caso corresponde al cálculo de las derivadas es con respecto a los pesos que se conectan a unidades ocultas. El cálculo de la derivada hasta estos pesos no se obtiene de manera directa, como el caso de las conexiones a las unidades de salida, por lo que la derivada se obtiene calculando

$$\frac{\partial E_p}{\partial w_{ij}} = \sum_k \frac{\partial E_p}{\partial y_k} \frac{\partial y_k}{\partial a_k} \frac{\partial a_k}{\partial y_j} \frac{\partial y_j}{\partial a_j} \frac{\partial a_j}{\partial w_{ij}}$$

donde k indexa a todas las nodos a los que se conecta la unidad j , por ahora suponemos que son las k unidades de salida. Simplificando la expresión, podemos ver que en los primeros dos términos estamos calculando los valores delta de las unidades de salida, de ahí el nombre de error de retropropagación.

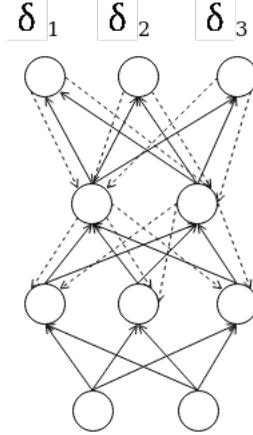


Figura 1.9: La retropropagación en una red de tres capas. Las líneas sólidas muestran el paso del feedforward y las líneas discontinuas muestran la propagación hacia atrás de los valores δ .

$$\begin{aligned}
 \frac{\partial E_p}{\partial w_{ij}} &= \sum_k \frac{\partial E_p}{\partial a_k} \frac{\partial a_k}{\partial y_j} \frac{\partial y_j}{\partial a_j} \frac{\partial a_j}{\partial w_{ij}} \\
 &= \sum_k \delta_k w_{jk} \frac{\partial y_j}{\partial a_j} \frac{\partial a_j}{\partial w_{ij}} \\
 &= \sum_k \delta_k w_{jk} f'(a_j) \frac{\partial a_j}{\partial w_{ij}} \\
 &= \delta_j y_i
 \end{aligned}$$

donde $\delta_j = \sum_k \delta_k w_{jk} f'(a_j)$. De manera general para calcular la derivada de la función de costo con respecto a cualquiera de los pesos tenemos

$$\frac{\partial E_p}{\partial w_{ij}} = \delta_j y_i$$

donde δ_j es el error de retropropagación de la unidad j , que es a la unidad a la que llega la arista con el peso w_{ij} y y_i es la salida de la unidad i que será la entrada del nodo j .

Actualización de los pesos

Después de obtener las derivadas, el siguiente paso es actualizar los pesos para disminuir el error. Como se dijo al principio, el término de retropropagación se refiere

al método eficiente para calcular las derivadas $\frac{\partial E}{\partial w}$ y al algoritmo de optimización que utiliza esas derivadas para ajustar los pesos y reducir el error.

La retropropagación como método de optimización es básicamente el descenso del gradiente. Sabemos que el gradiente negativo de E apunta a la dirección en la que E se decremente más rápido. Para minimizar E , los pesos son ajustados en la dirección del gradiente negativo. La regla para actualizar los pesos es

$$w_{ij} \leftarrow w_{ij} - \gamma \frac{\partial E}{\partial w_{ij}}$$

donde la *tasa de aprendizaje* $0 < \gamma$. Hay dos variaciones básicas para la actualización, el *modo por lotes* y el *online*.

- **Aprendizaje por lotes:** En este modo, cada patrón p es evaluado para obtener los términos de la derivada $\frac{\partial E_p}{\partial w}$; estos se suman para obtener la derivada total

$$\frac{\partial E}{\partial w} = \sum_p \frac{\partial E_p}{\partial w}$$

y sólo después de esto se actualizan los pesos. Los pasos son los siguientes:

1. Por cada patrón p en el conjunto de entrenamiento
 - Alimentar a la red con el patrón p y hacer la propagación hacia adelante para obtener la salidas de la red.
 - Calcular el error del patrón E_p y retropropagar para obtener las derivadas por patrón $\frac{\partial E_p}{\partial w}$.
2. Sumar todas las derivadas por patrón para obtener la derivada total.
3. Actualizar los pesos

$$w \leftarrow w - \gamma \frac{\partial E}{\partial w}$$

4. Repetir.

Cada paso a través de todo el conjunto de entrenamiento se llama *época*.

- **Aprendizaje online:** En este modo de aprendizaje, los pesos se actualizan después de que se presenta cada patrón. Generalmente, un patrón p se elige aleatoriamente y se presenta a la red. La salida se compara con el objetivo para ese patrón y los error son retropropagados para obtener una derivada $\frac{\partial E_p}{\partial w}$ para un solo patrón. Los pesos se actualizan inmediatamente después, usando el gradiente del error de un solo patrón. Los pasos son:

1. Elegir aleatoriamente un patrón p del conjunto de entrenamiento.

- Alimentar a la red con el patrón p y propagar hacia adelante para obtener las salidas de la red.
 - Calcular el error E_p y retropropagar para obtener las derivadas $\frac{\partial E_p}{\partial w}$.
2. Actualizar los pesos usando la derivada de un solo patrón
- $$w \leftarrow w - \gamma \frac{\partial E_p}{\partial w}$$
3. Repetir.

Retropagación en una forma matricial

En estructuras de redes prealimentadas con múltiples capas, es conveniente reescribir el método de retropropagación en una forma tal que las operaciones se simplifiquen a multiplicaciones de vectores por matrices, matrices por vectores, matrices por matrices y vectores por vectores. A continuación definiremos una red con dos capas, una oculta y una de salida, sin embargo, se puede ver que es posible generalizar para redes con más capas ocultas. Todas las operaciones que se realizan son con respecto a una muestra p , por lo que dependiendo de cómo se realice la actualización de los pesos, es como se deben de realizar las operaciones con las matrices de las derivadas que obtendremos al final.

Consideraremos una red con n unidades de entrada, k unidades ocultas y m unidades de salida. Hasta ahora la notación utilizada ha evitado que tratemos cada capa de una red por separado, pero ahora es necesario mantener un índice de la capa sobre la que se están haciendo los cálculo, por lo tanto se usa el superíndice (l) para referirnos a la capa l . Los pesos entre la unidad de entrada i y la oculta j se denotan por $w_{ij}^{(1)}$. El peso entre la unidad oculta i y la de salida j será $w_{ij}^{(2)}$.

Existen $n \times k$ pesos entre las unidades de entrada y las ocultas y $k \times m$ entre las ocultas y las de salida. Sea \mathbf{W}_1 la matriz de tamaño $n \times k$ cuyo componente $w_{ij}^{(1)}$ está en la i ésima fila y la j ésima columna. De manera similar definamos \mathbf{W}_2 como la matriz de $k \times m$ con elementos $w_{ij}^{(2)}$. El vector de entrada es de tamaño n , y lo definimos como $\mathbf{x} = (x_1, \dots, x_n)$.

Por ahora como función de activación usaremos a la sigmoide, por lo que la salida $y_j^{(1)}$ de la unidad es

$$y_j^{(1)} = f\left(\sum_i^n w_{ij}^{(1)} x_i\right) = \frac{1}{1 + e^{\sum_i^n w_{ij}^{(1)} x_i}}$$

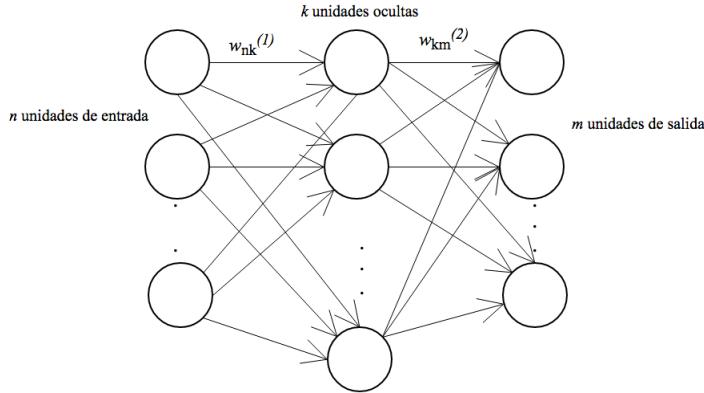


Figura 1.10: Arquitectura de la red propuesta para la notación matricial de la retropropagación.

Las salidas de la capa oculta se pueden obtener aplicando la función de activación a cada uno de los elementos que resulten de la multiplicación $\mathbf{x}\mathbf{W}_1$. El vector $\mathbf{y}^{(1)}$ cuyos componentes son las salidas de las unidades ocultas está dado por

$$\mathbf{y}^{(1)} = f(\mathbf{x}\mathbf{W}_1).$$

Las salidas de las unidades en la capa de salida se calculan usando el vector $\mathbf{y}^{(1)} = (y_1^{(1)}, \dots, y_k^{(1)})$. La salida de la red es un vector m dimensional $\mathbf{y}^{(2)}$, donde

$$\mathbf{y}^{(2)} = f(\mathbf{y}^{(1)}\mathbf{W}_2).$$

En el paso de feedforward, el vector \mathbf{x} alimenta a la red. Los vectores $\mathbf{y}^{(1)}$ y $\mathbf{y}^{(2)}$ son calculados. Además, en este paso se pueden obtener las derivadas evaluadas de las funciones de activación, que se pueden escribir en matrices diagonales \mathbf{D}_l . Para nuestra red de dos capas, \mathbf{D}_2 contiene las derivadas evaluadas de las funciones de activación de los nodos de salida, y \mathbf{D}_1 para las unidades ocultas. Por simplicidad, supusimos que f es la sigmoide y por lo tanto las derivadas son $f' = y(1 - y)$.

$$\mathbf{D}_2 = \begin{pmatrix} y_1^{(2)}(1 - y_1^{(2)}) & 0 & \dots & 0 \\ 0 & y_2^{(2)}(1 - y_2^{(2)}) & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & y_m^{(2)}(1 - y_m^{(2)}) \end{pmatrix}$$

y

$$\mathbf{D}_1 = \begin{pmatrix} y_1^{(1)}(1 - y_1^{(1)}) & 0 & \dots & 0 \\ 0 & y_2^{(1)}(1 - y_2^{(1)}) & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & y_k^{(1)}(1 - y_k^{(1)}) \end{pmatrix}$$

Para calcular los valores delta de la unidad de salida necesitamos las derivadas del error con respecto a las salidas. Definimos al vector \mathbf{e} con las derivadas de las desviaciones cuadráticas como

$$\mathbf{e} = \begin{pmatrix} -(t_1 - y_1^{(2)}) \\ -(t_2 - y_2^{(2)}) \\ \vdots \\ -(t_m - y_m^{(2)}) \end{pmatrix}$$

Para una unidad de salida $\delta_i^{(2)} = -(t_i - y_i^{(2)})y_i^{(2)}(1 - y_i^{(2)})$. Por lo tanto el vector m dimensional $\boldsymbol{\delta}^{(2)}$ que contiene todos los valores delta de la unidad de salida está dado por

$$\boldsymbol{\delta}^{(2)} = \mathbf{D}_2\mathbf{e}.$$

El vector de tamaño k de valores delta en la capa oculta es

$$\boldsymbol{\delta}^{(1)} = \mathbf{D}_1\mathbf{W}_2\boldsymbol{\delta}^{(2)}.$$

Después de calcular los vectores con los valores delta, es posible obtener las derivadas del error con respecto a los pesos. Las matrices con las derivadas del error con respecto los pesos de \mathbf{W}_2 y \mathbf{W}_1 , son respectivamente:

$$\nabla \mathbf{W}_2 = (\boldsymbol{\delta}^{(2)}\mathbf{y}^{(1)})^T$$

$$\nabla \mathbf{W}_1 = (\boldsymbol{\delta}^{(1)}\mathbf{x})^T$$

$$\nabla \mathbf{W}_2 = \begin{pmatrix} \frac{\partial E}{w_{11}^{(2)}} & \dots & \frac{\partial E}{w_{1m}^{(2)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial E}{w_{k1}^{(2)}} & \dots & \frac{\partial E}{w_{km}^{(2)}} \end{pmatrix}$$

$$\nabla \mathbf{W}_1 = \begin{pmatrix} \frac{\partial E}{w_{11}^{(1)}} & \cdots & \frac{\partial E}{w_{1k}^{(1)}} \\ \vdots & \ddots & \vdots \\ \frac{\partial E}{w_{n1}^{(1)}} & \cdots & \frac{\partial E}{w_{nk}^{(1)}} \end{pmatrix}$$

Es fácil generalizar estas ecuaciones para l capas de unidades de cómputo. Asumamos que la matriz de conexión entre la capa i e $i+1$ está denotada por \mathbf{W}_{i+1} . El vector $\delta^{(l)}$ de la capa de salida es entonces

$$\delta^{(l)} = \mathbf{D}_l \mathbf{e}$$

El vector $\delta^{(i)}$ hasta la i ésima capa se define recursivamente

$$\delta^{(i)} = \mathbf{D}_i \mathbf{W}_{i+1} \delta^{(i+1)} \text{ para } i = 1, \dots, l-1$$

o de manera alternativa

$$\delta^{(i)} = \mathbf{D}_i \mathbf{W}_{i+1} \dots \mathbf{W}_{l-1} \mathbf{D}_{l-1} \mathbf{W}_l \mathbf{D}_l \mathbf{e}$$

Las correcciones para las matrices de pesos se calculan de la misma manera que para las dos capas de las unidades de cómputo.

1.2.3. Aprendizaje profundo

El *aprendizaje profundo* permite a las computadoras construir conceptos complejos a partir de conceptos más simples. La figura 1.11 muestra como un sistema de aprendizaje profundo puede representar el concepto de la imagen de una persona al combinar conceptos más simples, tales como esquinas y contornos, que a su vez están definidos en términos de los bordes. El ejemplo por excelencia de un modelo de aprendizaje profundo es una *red neuronal feedforward profunda*, donde lo profundo significa que tiene dos o más capas ocultas. Una red neuronal es simplemente una función matemática que mapea algún conjunto de valores de entrada en valores de salida. La función está formada por una composición de funciones más simples. Podemos suponer que al aplicar cada función matemática se obtiene una nueva representación de la entrada.

Remark. *El aprendizaje profundo es un tipo de aprendizaje automático poderoso y flexible que aprende una representación del mundo como una jerarquía anidada de conceptos, donde cada concepto está definido a partir de conceptos más simples, y representaciones más abstractas calculadas en términos de menos abstractas.*

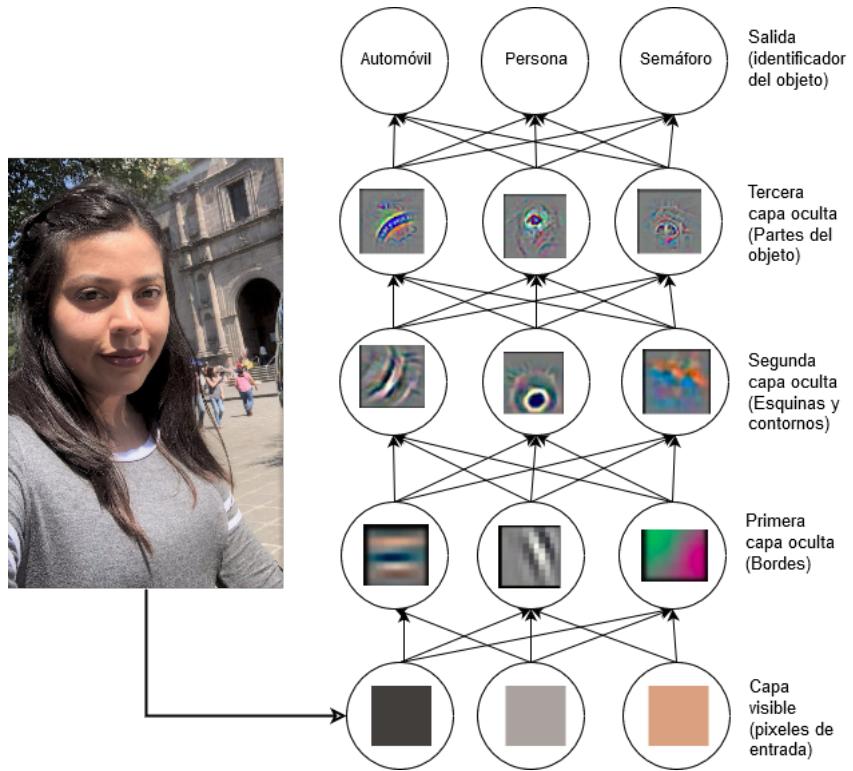


Figura 1.11: Ejemplo de un modelo de aprendizaje profundo. El aprendizaje profundo resuelve el problema de mapear un conjunto de pixeles al identificador de un objeto separando este mapeo complejo en mapeos simples anidados, cada uno descrito por una capa diferente en el modelo. La entrada es presentada en la capa visible, llamada así porque contiene las variables que somos capaces de observar. Después una serie de capas ocultas extraen de la imagen características cada vez más abstractas. Estas capas se llaman *ocultas* porque sus valores no son dados en los datos; en su lugar, el modelo debe determinar qué conceptos son útiles para explicar las relaciones de los datos observados. Dados los pixeles, la primera capa puede fácilmente identificar bordes comparando el brillo entre vecindades de pixeles. Dada la descripción de bordes de la primera capa, la segunda capa puede con facilidad buscar esquinas y contornos que son reconocibles como colecciones de bordes. Dada el resultado de la segunda capa oculta, la tercera capa detecta partes enteras de objetos específicos, al encontrar colecciones específicas de contornos y esquinas. Finalmente, esta descripción de la imagen en términos de partes de objetos puede ser usada para reconocer objetos presentes en la imagen.

1.2.4. Redes neuronales convolucionales

Remark. *Las redes neuronales convolucionales o CNN (por sus siglas en inglés), son un tipo especializado de red neuronal para procesamiento de datos que tienen una topología parecida a una cuadrícula. El nombre de red convolucional indica que la red ocupa una operación matemática llamada convolución. Las redes neuronales convolucionales son simplemente redes neuronales que usan la convolución en lugar de la multiplicación de matrices en al menos una de sus capas.*

Las redes convolucionales son una categoría dentro de las redes neuronales profundas, que han demostrado ser muy efectivas en áreas como el reconocimiento y clasificación de imágenes. Debido a los excelentes resultados de las CNN en estas áreas, nos enfocaremos en estas redes para tareas de visión computacional.

Remark. *La correlación es una operación que funciona recorriendo una imagen y aplicando un filtro a cada pixel. El filtro se denota como un kernel. Primero el kernel se llena con números llamados coeficientes de kernel. Estos coeficientes ponderan el valor del pixel que cubren y la salida de la correlación es la suma de los valores de los pixeles ponderados. Matemáticamente la correlación de la imagen I con el kernel K se puede expresar como sigue:*

$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(i + m, j + n)K(m, n). \quad (1.3)$$

La correlación está asociada con el término de *convolución* y ambos se utilizan en el contexto de procesamiento de imágenes. La convolución únicamente difiere en la manera en que el kernel se aplica a la imagen. Formalmente la convolución se define como:

$$S(i, j) = \sum_m \sum_n I(i - m, j - n)K(m, n).$$

Comparando esta ecuación con la de correlación (ecuación 1.3), podemos ver que las únicas diferencias son los signos negativos. La interpretación de esto es que el kernel está rotado 180 grados antes de hacer la correlación.

Cuando se aplica un filtro para suavizar una imagen, encontrar bordes, etcétera, el proceso se denota como convolución incluso si se implementa como una correlación. Por lo tanto, por conveniencia llamaremos a ambas operaciones convolución.

En la figura 1.12 para calcular cada valor en la salida, el kernel se mueve con un paso de tamaño uno sobre el eje horizontal y vertical. Este paso se conoce como *zancada* o *stride* (en inglés). El valor de la zancada puede ser diferente de uno,

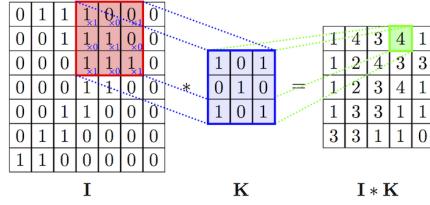


Figura 1.12: Convolución entre una imagen I y un kernel K sin rotarlo 180 grados.

según convenga. El tamaño de $I * K$ para una imagen I de tamaño $h \times w$ y un filtro de tamaño $f \times f$ depende también de la zancada s .

$$h' = \lfloor \frac{h - f - s}{s} \rfloor$$

$$w' = \lfloor \frac{w - f - s}{s} \rfloor$$

donde h' y w' son las dimensiones del resultado. Se puede notar que entre más grande sea el valor de s el tamaño de $I * K$ disminuye. Sin embargo, en algunas aplicaciones es necesario conservar el tamaño de la entrada. Esto último se logra aplicando un *borde de ceros* a la imagen de entrada. Al añadir el borde de ceros las dimensiones el tamaño del resultado de la convolución se incrementa. Si p denota el incremento sobre cada dimensión de I , el tamaño de h' y w' se calculan de la siguiente forma:

$$h' = \lfloor \frac{h - f - s + p}{s} \rfloor$$

$$w' = \lfloor \frac{w - f - s + p}{s} \rfloor$$

La convolución aprovecha dos ideas importantes que pueden ayudar a mejorar un sistema de aprendizaje automático: *interacciones escasas* y *compartición de parámetros*.

Las capas de las redes neuronales tradicionales usan la multiplicación de matrices de parámetros con un parámetro separado entre cada unidad de entrada y cada unidad de salida. Esto significa que cada unidad de salida interactúa con cada unidad de entrada. Las redes convolucionales, sin embargo, tienen *interacción escasa* (también conocida como conectividad escasa o pesos escasos). Esto se logra haciendo que el kernel sea de menor tamaño que la entrada. Por ejemplo, cuando se procesa una imagen, la imagen de entrada puede tener miles o millones de pixeles, pero podemos

detectar pequeñas y significativas características como bordes con kernels que sólo ocupan decenas o cientos de pixeles. En consecuencia se puede ver que se almacenan menos parámetros, lo que reduce los requisitos de memoria del modelo y mejora la eficiencia estadística. Además significa que el cálculo de las salidas requiere menos operaciones.

La *compartición de parámetros* se refiere a usar el mismo parámetro para más de una función en un modelo. En una red neuronal tradicional, cada elemento de la matriz de pesos se usa exactamente una vez cuando se calcula la salida de una capa. Se multiplican por un elemento en la entrada y nunca se vuelve a visitar. En una CNN, cada elemento del kernel se usa en cada posición de la entrada. La compartición de parámetros indica que más que aprender un conjunto de parámetros por separado para cada posición, aprendemos sólo un conjunto.

Sabemos que cada kernel puede verse como una cuadrícula de números también llamados pesos y que éstos se aprenden durante el entrenamiento de la CNN. El proceso de aprendizaje involucra la inicialización de los pesos del kernel al comenzar el entrenamiento. Posteriormente, dados los pares de entrada-salida, los pesos son ajustados en varias iteraciones durante el proceso de aprendizaje.

Remark. *Una capa típica en una CNN consiste de tres etapas. En la primera etapa, la capa realiza diferentes convoluciones en paralelo para producir una colección de mapas de características. En la segunda etapa cada mapa de características se pasa por una función de activación no lineal. La función de activación más común es el rectificador o ReLU definido como $g(z) = \max(0, z)$. Esta etapa también se conoce como detector. En la tercera etapa utilizamos una función de agrupación o pooling para modificar la salida de la capa aún más.*

Remark. *Una función de pooling reemplaza la salida de la red en cierta ubicación con una estadística resumida de las salidas vecinas. Por ejemplo, la operación de agrupamiento máximo retorna el valor máximo dentro de una vecindad rectangular.*

El agrupamiento ayuda a hacer que la representación sea más invariante a pequeñas traslaciones de la entrada. La *invarianza a las traslaciones* significa que si trasladamos la entrada por una pequeña cantidad, los valores de la mayoría de las salidas agrupadas no cambian. La invarianza a traslaciones locales puede ser muy útil cuando nos importa si una característica está presente y no exactamente donde.

A las capas que pasan por las tres etapas las llamaremos *capas de convolución*, aunque para algunos autores, cada etapa es una capa diferente, la primera es de convolución, la segunda de pooling y la tercera de no linealidad. Por simplicidad elejimos el término de capa de convolución.

1.2.5. TensorFlow

TensorFlow es una biblioteca de código abierto para computo numérico. Fue desarrollada originalmente por investigadores e ingenieros del equipo de Google Brain, con el propósito de motivar la investigación de arquitecturas profundas.

TensorFlow es una interfaz para expresar algoritmos de aprendizaje automático y una implementación para ejecutar tales algoritmos. Un cálculo expresado en TensorFlow puede ser ejecutado con ningún o con mínimos cambios en una amplia variedad de sistemas desde dispositivos móviles como teléfonos o tabletas hasta sistemas distribuidos a gran escala de cientos de máquinas y cientos de dispositivos con unidades de procesamiento gráfico (GPU). El sistema es flexible y puede usarse para expresar una gran variedad de algoritmos, incluyendo algoritmos de entrenamiento e inferencia para modelos de redes neuronales profundas y ha sido usado para llevar a cabo investigaciones en áreas como la visión computacional, robótica, procesamiento de lenguaje natural, etcétera.

Remark. *Los cálculos en TensorFlow son descritos por una gráfica dirigida, que está compuesta por un conjunto de nodos y aristas. La gráfica representa un cálculo de flujo de datos (un modelo de programación para cómputo paralelo).*

A menudo los clientes contruyen la gráfica de cómputo usando uno de los lenguajes de frontend soportados (C++ o Python). Todos los ejemplos que se muestran en esta sección están desarrollados en Python.

En una gráfica de TensorFlow los nodos representan instancias de *operaciones*, tiene cero o más entradas y cero o más salidas. Los valores que fluyen sobre las aristas en la gráfica son *tensores*. Estos últimos son arreglos de dimensiones arbitrarias donde el tipo de sus elementos se especifica o infiere durante la construcción de la gráfica.

Una *operación* tiene un nombre y representa un cálculo abstracto, por ejemplo, una multiplicación de matrices o una suma.

Los programas clientes interactúan con el sistema de TensorFlow creando una *Sesion*. El objeto Session es una interfaz para la gráfica de cómputo, es quien permite su ejecución a través del su operacion primaria *Run*, la cual toma un conjunto de nodos de la gráfica que necesitan ser calculados, así como un conjunto de tensores que necesitan alimentar a la gráfica para poder obtener las salidas de ciertos nodos. TensorFlow puede calcular la cerradura transitiva de todos los nodos que deben ser ejecutados para poder calcular las salidas que fueron solicitadas en los argumentos de *Run*.

Tabla 1.4: Algunos de los tipos de operaciones en TensorFlow.

Categoría	Ejemplo
Operaciones matemáticas a nivel de elementos	Add, Sub, Mul, Div, Exp, Log, Greater, Less, Equal, ...
Operaciones con arreglos	Concat, Slice, Split, Constant, Rank, Shape, Shuffle, ...
Operaciones con matrices	MatMul, MatrixInverse, MatrixDeterminant, ...
Construcción de redes neuronales	SoftMax, Sigmoid, ReLU, Convolution2D, MaxPool
Operaciones de puntos de control	Save, Restore

Variables

Cuando construimos un modelo de aprendizaje en TensorFlow utilizamos *variables* para representar los parámetros del modelo. Las variables de TensorFlow son buffers en memoria que contienen tensores; pero a diferencia a los tensores normales que sólo son instanciados cuando una gráfica se ejecuta y que inmediatamente después se eliminan, las variables se mantienen durante múltiples ejecuciones de una gráfica. Como resultado, las variables de TensorFlow tienen las siguientes tres propiedades:

- Las variables deben inicializarse explícitamente antes de que una gráfica se utilice por primera vez.
- Podemos utilizar métodos de optimización como el del descenso del gradiente para modificar las variables después de cada iteración para ajustar los parámetros óptimos del modelo.
- Podemos guardar en el disco los valores de las variables y restaurarlos para su uso posterior.

Crear e inicializar una variable es muy simple, únicamente utilizamos la clase **Variable** dentro de la biblioteca de TensorFlow. Comencemos por inicializar una variable que describe los pesos entre dos capas en una red neuronal.

```
weights = tf.Variable(tf.random_normal([300, 200], stddev=0.5),
name="weights")
```

`tf` es un alias para el módulo `tensorflow`. `tf.Variable` recibe dos argumentos; el primero, `tf.random_normal`, que es una operación que produce un tensor inicializado usando una distribución normal con una desviación estándar de 0.5 y media

0. El tamaño del tensor es de 300×200 , lo que significa que los pesos conectan a una capa de 300 neuronas a una de 200. El segundo parámetro es un identificador único que nos permite referirnos a un nodo en específico en la gráfica de cómputo.

Por defecto las variables son entrenables, lo que quiere decir que cuando utilizamos algo como el descenso del gradiente, las variables automáticamente cambian sus valores.

PlaceHolder

Los modelos de aprendizaje necesitan una entrada ya sea para el entrenamiento y la evaluación. Una variable no es suficiente porque sólo se inicializa una vez. Necesitamos un componente que llenemos cada que la gráfica de cómputo se ejecuta. TensorFlow usa una estructura llamada *placeholder*.

Un placeholder es instanciado como sigue y puede ser utilizado en operaciones como los tensores y variables en TensorFlow.

```
x = tf.placeholder(tf.float32, name="x", shape[None, 1024])
W = tf.Variable(tf.random_uniform([784,10], -1, 1), name="W")
multiply = tf.matmul(x, W)
```

En el ejemplo definimos un placeholder donde `x` representa un lote de datos de tipo `float32`. Podemos notar que `x` tiene 1024 columnas, lo que significa que cada patrón de entrada tiene una dimensión de 1024. Se puede notar también que `x` tiene un número indefinido de filas. Esto significa que `x` puede ser inicializado con un número arbitrario de muestras. Aunque podemos multiplicar cada muestra por separado con `W`, al expresar un lote completo como un tensor tenemos permitido calcular los resultados de todas los patrones de entrada en paralelo. El resultado de la fila `i` del tensor de multiplicación corresponde a multiplicar `W` con el patrón `i`.

Así como las variables se inicializan la primera vez que se construye la gráfica, los placeholder necesitan llenarse cada vez que se ejecuta la gráfica o una subgráfica de cómputo.

Ejemplo de una red neuronal en TensorFlow

En este apartado se describe la creación de un modelo de aprendizaje muy simple, una red neuronal, en TensorFlow para el cálculo de la función **XOR**.

Como estamos en el contexto del aprendizaje automático supervisado, antes de definir la gráfica necesitamos un conjunto de datos. Nuestro conjunto de datos está dado por la tabla 1.5.

Tabla 1.5: Definición de la función XOR.

X_1	X_2	$Y = X_1 \oplus X_2$
0	0	0
1	0	1
0	1	1
1	1	0

La arquitectura de nuestra red es la siguiente: una capa de entrada con dos unidades de entrada X_1 y X_2 . Una capa oculta con dos unidades, que tienen como función de activación la sigmoide. Por último una unidad de salida con la función sigmoide. De manera gráfica esto se puede ver en la figura 1.13.

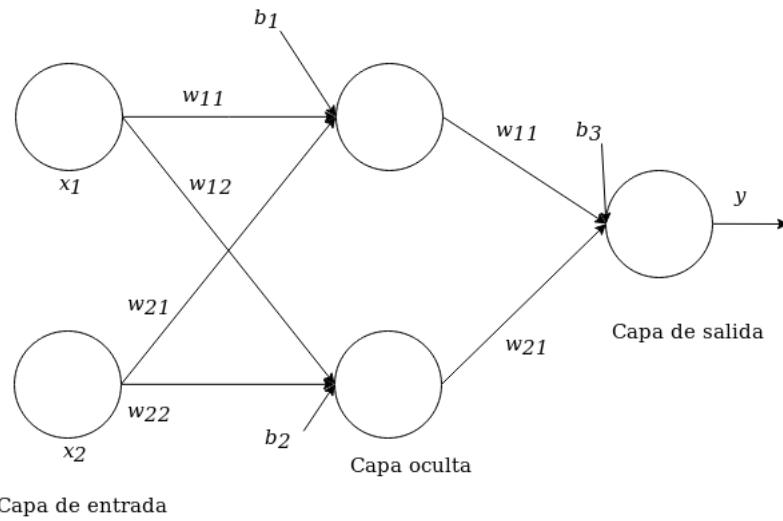


Figura 1.13: Arquitectura de la red neuronal para calcular la función \oplus .

Antes de comenzar con el código conviene dibujar la gráfica de cómputo que deseamos ejecutar. La gráfica de cómputo está escrita en la figura 1.14. Los círculos representan o variables o placeholders excepto para la salida predicha y' que es el resultado de una operación. Los placeholders son nuestras entradas X y las salidas correctas u objetivos y . Las variables son los pesos W_i y los sesgos b_i . Los rectángulos son las operaciones. En una red neuronal las únicas operaciones son la multiplicación y suma de tensores, la aplicación de la función de activación y un método de optimización como el descenso del gradiente para minimizar el error dado por una función de costo, en nuestro caso la suma de errores cuadráticos (SSE). Durante la ejecución de la gráfica las variables cambiarán su valor automáticamente a los valores óptimos que minimicen el error.

Un programa en TensorFlow se compone de dos partes, la creación de la gráfica y su ejecución.

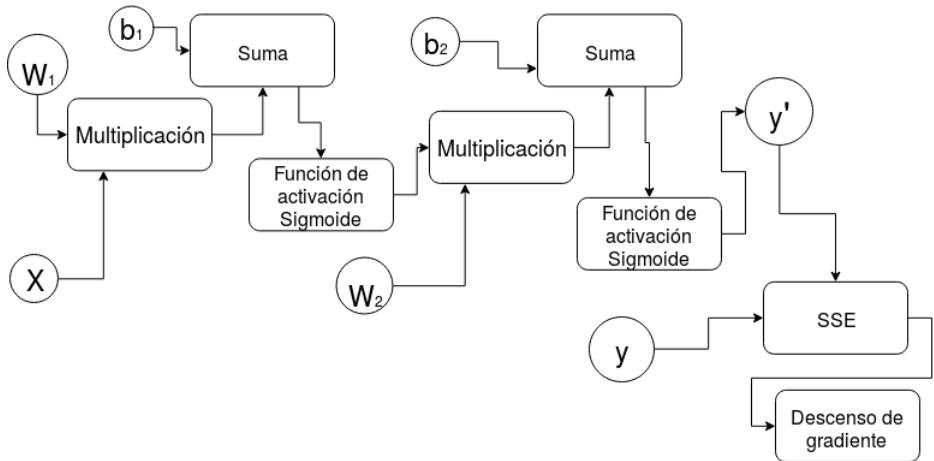


Figura 1.14: La gráfica de TensorFlow que representa a la arquitectura de la red para el cálculo de la función \oplus .

Antes que nada como para cualquier biblioteca, se importa tensorflow.

```
import tensorflow as tf
```

El conjunto de datos se puede definir en dos listas, una con los datos de entrada y la otra con sus respectivas salidas.

```
datos_de_entrada = [[0., 0.], [1., 0.], [0., 1.], [1., 1.]] # una lista
                    ↪con los diferentes pares (x1, x2)
y = [[0.], [1.], [1.], [0.]] #los objetivos o etiquetas
```

Definición de la gráfica de cómputo

Se definen los placeholders que se llenan con los patrones de entrada y sus etiquetas.

```
entrada_de_la_red = tf.placeholder(tf.float32, shape=[4, 2]) # el
                    ↪placeholder que se llena con los datos_de_entrada
salidas_correctas = tf.placeholder(tf.float32, shape=[4, 1]) # el
                    ↪placeholder para las etiquetas
```

Después se crean las variables para los pesos y sesgos de la red, los parámetros que se aprenden.

```
weights_layer_one = tf.Variable(tf.random_normal([2, 2])) # valores
                    ↪aleatorios de la distribución normal media=0.0 desviación estándar=1.0
weights_layer_two = tf.Variable(tf.random_normal([2, 1])) # media=0.0
                    ↪desviación estándar=1.0
```

```

bias_layer_one = tf.Variable(tf.random_normal([2])) # media=0.0 ↴
↳ desviación estándar=1.0
bias_layer_two = tf.Variable(tf.random_normal(())) # media=0.0 desviación ↴
↳ estándar=1.0

```

Necesitamos las operaciones que aplican la función de activación del resultado de la suma del producto de los pesos con las entradas más los sesgos. Estas operaciones generan la capa oculta. La salida de la red se compone de las mismas operaciones pero sus entradas son las salidas de la capa oculta.

```

output_layer_one = tf.sigmoid(tf.matmul(entrada_de_la_red, weights_layer_
↳ one) + bias_layer_one)
network_output = tf.sigmoid(tf.matmul(output_layer_one, weights_layer_
↳ two) + bias_layer_two)

```

Se define la función de costo que se desea minimizar, la SSE.

```
error = tf.reduce_sum(tf.square(salidas_correctas - network_output))
```

Elegimos el optimizador de la red para minimizar el error cambiando los parámetros (pesos y sesgos), ocupamos el descenso del gradiente (retropropagación).

```

optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.1)
train = optimizer.minimize(error)

```

Ejecución de la gráfica de cómputo

Finalmente se ejecuta la gráfica de TensorFlow. Se crea una instancia de `Session` e iteramos ejecutando el método `run` que ejecuta el entrenamiento del modelo.

```

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer()) #inicializa las Variables
    epochas = 10000
    print("Salida de la red antes del entrenamiento: {}".format(sess.
        ↳ run(network_output, feed_dict={entrada_de_la_red : datos_de_entrada, ↴
        ↳ salidas_correctas : y})))

    for i in range(epochas):
        sess.run(train, feed_dict={entrada_de_la_red: datos_de_entrada, ↴
        ↳ salidas_correctas: y})

    print("Salida de la red después del entrenamiento: {}".format(sess.
        ↳ run(network_output, feed_dict={entrada_de_la_red : datos_de_entrada, ↴
        ↳ salidas_correctas : y})))

```

La salida del programa es la siguiente:

```
Salida de la red antes del entrenamiento:  
[[0.3293548 ]  
[0.32501128]  
[0.20175578]  
[0.20124543]]  
Salida de la red después del entrenamiento:  
[[0.03212966]  
[0.9716061 ]  
[0.9716902 ]  
[0.02967284]]
```

1.3. Cómputo en la nube

Remark. Según el Instituto Nacional de Estándares y Tecnología (NIST por sus siglas en inglés) de Estados Unidos; el cómputo en la nube se define como un modelo que hace posible el acceso ubicuo, conveniente y bajo demanda a un grupo de recursos informáticos configurables (p.ej. servidores, almacenamiento, redes, aplicaciones, y servicios) a través de una red donde pueden ser rápidamente suministrados y lanzados con un mínimo esfuerzo de mantenimiento o interacción con los proveedores de servicios.

El cómputo en la nube también se puede definir como un *estilo de cómputo en el cual se brindan recursos como servicios, bajo demanda, a través de internet*. Estos recursos se escalan dinámicamente y a menudo son virtualizados.

Con esta tecnología los usuarios, casi con cualquier dispositivo, pueden acceder a programas, servicios de alojamiento de archivos, plataformas para desarrollo de aplicaciones, etcétera; mediante internet, usando servicios ofrecidos por proveedores de cómputo en la nube. Las ventajas del cómputo en la nube incluyen reducción de costos, alta disponibilidad y la fácil escalabilidad.

1.3.1. Capas del cómputo en la nube

El cómputo en la nube puede verse como una colección de servicios, que puede ser presentada como una arquitectura de cómputo en la nube en capas. Estas tres capas son: *Software as a Service (SaaS)*, *Infrastructure as a Service (IaaS)* y *Platform as a Service (PaaS)*.

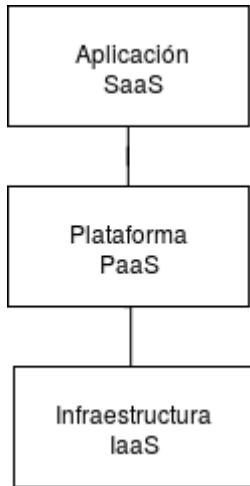


Figura 1.15: Arquitectura en capas del cómputo en la nube

Software as a Service (SaaS)

El *Software como un Servicio* se refiere simplemente a software que se entrega bajo demanda para su uso. Pongamos el siguiente ejemplo; antes si alguien necesitaba un programa para edición de textos tenía que ir a una tienda, comprar algún disco, e instalarlo en su computadora. Tiempo después una nueva actualización salía al mercado, entonces se repetía el proceso. Con SaaS, sólo es necesario ingresar desde el navegador a un programa alojado en alguna parte. No hay instalación, no hay actualización.

Infrastructure as a Service (IaaS)

La *Infraestructura como un servicio* hace referencia a recursos de cómputo como un servicio. Esto incluye computadoras virtualizadas con garantía de poder de procesamiento y ancho de banda reservado para almacenamiento y acceso a internet. Los servicios más comunes de la IaaS son:

- Alojamiento
- Balance de carga
- Conectividad de red pública y privada
- Firewalls
- Almacenamiento

Platform as a Service (PaaS)

Esta se encuentra en medio de las otras dos, con IaaS en la parte de abajo y SaaS en la parte más alta. Se encarga de proveer todo lo necesario para ejecutar un lenguaje en específico o un *stack de soluciones*. Es similar a IaaS pero incluye un sistema operativo y servicios requeridos para una aplicación en específico.

1.3.2. Servicios en la nube

Existen tres categorías de servicios en la nube. La primera es el servicio en la nube SaaS, donde la aplicación entera corre en la nube. El cliente utiliza un navegador para acceder a la aplicación. Un ejemplo de SaaS es Office 365.

El otro tipo de servicio es en el que la aplicación corre del lado del cliente, sin embargo, accede a algunas funcionalidad y servicios provistos en la nube. Un ejemplo es Spotify. La aplicación móvil reproduce la música, mientras que el servicio en la nube es utilizado para descargar nuevas canciones.

El último tipo es una plataforma en la nube para crear aplicaciones, usada por desarrolladores. Los desarrolladores crean una nueva aplicación como SaaS usando una plataforma en la nube, un ejemplo es Cloud9.

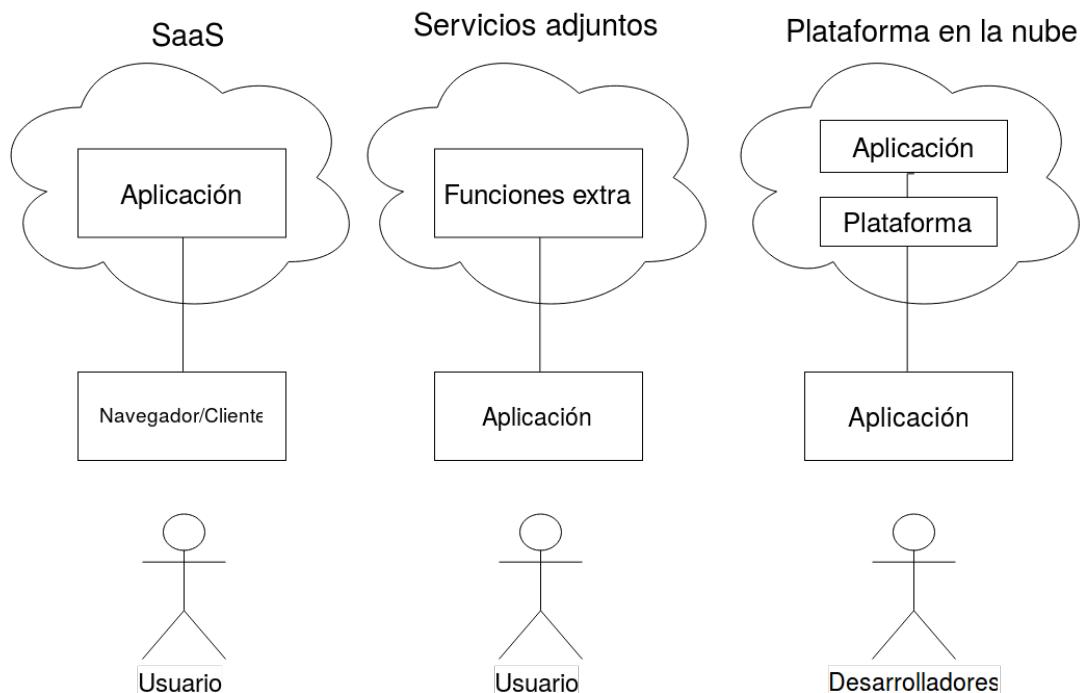


Figura 1.16: Las categorías de los servicios en la nube.

1.3.3. Tipos de cómputo en la nube

Hay tres tipos de cómputo en la nube:

- **Nube pública**
- **Nube privada**
- **Nube híbrida**

En la nube pública, los recursos de cómputo son proporcionados dinámicamente a través de internet por medio de aplicaciones web o servicios web brindados por terceros. Las nubes públicas son ejecutadas por terceros y la aplicaciones de diferentes consumidores probablemente se mezclan en los servidores en la nube, los sistemas de almacenamiento y las redes.

La nube privada es el cómputo en la nube sobre redes privadas. Están construidas exclusivamente para un cliente, proporcionando un control total sobre los datos, la seguridad y la calidad del servicio. La nube privada puede ser mantenida por la organización que la ocupará o por terceros.

Una nube híbrida combina múltiples modelos de nubes privadas y públicas. Añaden la complejidad de determinar como distribuir aplicaciones entre ambos tipos de nubes.

1.3.4. Cómputo en la nube vs servicios en la nube

En esta sección se presentan dos tablas que muestran las diferencias y los principales atributos del cómputo en la nube y los servicios en la nube. El cómputo en la nube consiste de las tecnologías que permiten los servicios en la nube. Los atributos clave del cómputo en la nube y de los servicios en la nube se muestran en las tablas Tabla 1.3.4 y Tabla 1.3.4, respectivamente.

Tabla 1.6: Atributos principales del cómputo en la nube

Atributo	Descripción
Sistemas de infraestructura	Incluye servidores, almacenamiento, y redes que pueden escalarse como el usuario lo demande.
Aplicación	Provee una interfaz de usuario basada en la web., servicios web de API, y una amplia variedad de configuraciones.
Desarrollo aplicaciones y lanzamiento de software	Soporta el desarrollo e integración de una aplicación de software en la nube.
Software de mantenimiento para sistemas y aplicaciones	Soporta una provisión rápida de autoservicio y configuración, además de un monitoreo de uso.
Redes IP	Conectan a los usuarios finales a la nube y los componentes de la infraestructura.

Tabla 1.7: Atributos principales de los servicios en la nube

Atributo	Descripción
<i>Offsite.</i> Proveedor externo.	En la ejecución en la nube, se asume que hay terceros que proveen servicios. También existe la posibilidad de una entrega de servicios en la nube de manera interna.
Acceso vía internet	Los servicios se acceden por medio de una red universal basada en un estándar. Se puede incluir las opciones de seguridad y calidad del servicio.
Habilidades de IT nulas o mínimas requeridas	Hay una especificación simplificada de requerimientos.
Aprovisionamiento	Esto incluye la solicitud de autoservicio, una implementación casi en tiempo real, y una escalamiento dinámico.
Precio	El precio está basado en la capacidad de uso.
Interfaz de usuario	La interfaz de usuario incluye navegadores para una variedad de dispositivos.
Interfaz del sistema	Las interfaces del sistema están basadas en API web de servicios brindando un framework estándar para acceso e integración entre los servicios en la nube.
Recursos compartidos	Los recursos son compartidos entre los usuarios de los servicios en la nube; sin embargo, a través de opciones configuración, existe la posibilidad de personalizar.

1.3.5. Tecnologías que hacen posible el cómputo en la nube

Algunas de las tecnologías claves que permiten el cómputo en la nube se describen de manera muy general y breve a continuación.

Virtualización

La ventaja del cómputo en la nube es la habilidad de virtualizar y compartir recursos entre diferentes aplicaciones con el objetivo de utilizar mejor a los servidores. En el cómputo no en la nube, por ejemplo, tres plataformas existen para tres aplicaciones diferentes corriendo en su propio servidor. En la nube, los servidores pueden ser compartidos o virtualizados, para distintos sistemas operativos y aplicaciones resultando en menos servidores.

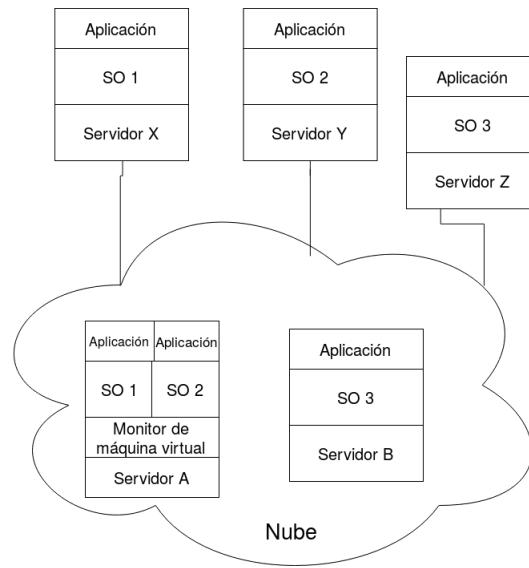


Figura 1.17: Un ejemplo de virtualización: en un cómputo que no esté en la nube se necesitan tres servidores, en la nube sólo se usan dos servidores.

Servicios Web y Arquitectura Orientada a Servicios

Los servicios en la nube están diseñados típicamente como servicios web (sistemas de software diseñados para soportar la interacción interoperable máquina a máquina a través de una red), que siguen los estándares de la industria como SOAP, REST, entre otros. La arquitectura orientada a servicios (SOA por sus siglas en inglés) organiza y maneja los servicios web dentro de las nubes. Una SOA también incluye

un conjunto de servicios en la nube, que están disponibles sobre varias plataformas distribuidas.

Flujo de servicio y flujos de trabajo

El concepto de flujo de servicio y flujos de trabajo se refiere a una vista integrada de actividades basadas en servicios provistas por la nube. Los flujos de trabajo se han vuelto una de las áreas importantes de investigación en el campo de los sistemas de bases de datos y sistemas de información.

1.3.6. Características del cómputo en la nube

El cómputo en la nube brinda un número de nuevas características a otros paradigmas de computación.

- Escalabilidad y servicios bajo demanda. El cómputo en la nube provee recursos y servicios para usuarios bajo demanda. Los recursos son escalables sobre varios centros de datos.
- Interfaz centrada en el usuario. Las interfaces de la nube son independientes de la ubicación y pueden ser accedidas a través de servicios web o navegadores.
- Calidad de servicio garantizada. El cómputo en la nube garantiza calidad de servicio para los usuarios en términos de desempeño de hardware, ancho de banda y capacidad de la memoria.
- Sistemas autónomos. Los sistemas de cómputo en la nube son sistemas autónomos administrados de manera transparente a los usuarios. Sin embargo, el software e información dentro de las nubes pueden reconfigurarse automáticamente y consolidarse en una plataforma simple dependiendo de las necesidades del usuario.
- Precio. Los usuarios pagan por los servicios y capacidad que necesitan.

1.3.7. Robótica en la nube

Remark. *Un robot en la nube se define de manera muy general como cualquier robot o sistema de automatización que depende ya sea de datos o código enviado a través de una red que soporta su operación, es decir, donde no todo el cómputo, y memoria está integrada en un solo sistema.*

Debido a factores como la latencia de la conexión, calidad de servicio variable, etcétera; un robot en la nube a menudo tiene cierta capacidad para procesamiento local para respuestas de baja latencia y durante periodos donde el acceso a una red no esté disponible o no sea confiable.

Algunos beneficios potenciales obtenidos de la nube son:

- **Big data:** Acceso a bibliotecas remotas de imágenes, mapas, trayectorias, y datos de objetos.
- **Cómputo en la nube:** Acceso de cómputo bajo demanda en paralelo en un grid de cómputo para análisis estadístico, aprendizaje y planeación de movimiento.
- **Aprendizaje colectivo de robots:** Robots compartiendo sus trayectorias, políticas de control, y salidas.
- **Cómputo humano:** Acceso a *crowdsourcing* (colaboración abierta distribuida) para utilizar la experiencia y habilidad humana en el análisis de imágenes, o en la recopilación de datos. Y acceso a *call centers* que no es más que la operación remota por humanos.

1.4. Arquitecturas de Servicios de Transferencia de Estado Representacional

1.4.1. Protocolo de Transferencia de Hipertexto

Remark. *El Protocolo de Transferencia de Hipertexto o HTTP (por sus siglas en inglés) es un protocolo de la capa de aplicación para sistemas de información hipermédia distribuidos y colaborativos.*

Los navegadores web, servidores y todas las aplicaciones web relacionadas se comunican entre ellas a través de este protocolo. Es la base de la World Wide Web. Cada vez que se realiza una transacción en la web, HTTP es invocado.

El contenido de la Web reside en servidores web. Los servidores web “hablan” el protocolo HTTP, por lo que son llamados servidores HTTP. Estos servidores HTTP almacenan datos de Internet y proveen estos datos cuando son solicitados por los clientes HTTP. Los clientes hacen peticiones HTTP a los servidores, y los servidores regresan los datos solicitados en respuestas HTTP. Juntos, los clientes y servidores HTTP conforman los componentes básicos de la World Wide Web.

Los servidores web organizan y almacenan los *recursos web*. Un recurso web es la

fuente del contenido web. La forma más simple de un recurso web es un archivo estático en el sistema de archivos del servidor web. Estos archivos pueden contener cualquier cosa: archivos de texto, imágenes en formato JPEG, archivos de video MP4, etcétera. Sin embargo, los recursos no son necesariamente archivos estáticos. Pueden ser programas que generen contenido cuando se les solicite. Estos recursos dinámicos pueden generar contenido de acuerdo a la identidad de un usuario, hora del día, etc. Estos te pueden mostrar imágenes en vivo de una cámara, acciones, o hacer búsquedas en bases de datos. En resumen, un recurso es cualquier tipo de fuente de contenido.

Una *transacción* en HTTP consiste de un comando de *solicitud* o *peticIÓN* (enviada desde un cliente a un servidor), y una *respuesta* como resultado (enviado del servidor al cliente). Esta comunicación se lleva a cabo con bloques de datos, con un formato definido, llamados *mensajes HTTP*.

Métodos

El protocolo HTTP soporta varias peticiones diferentes, conocidas como métodos HTTP. Cada mensaje de solicitud HTTP tiene un método. El método le dice al servidor que acción realizar (buscar una página web, eliminar un archivo, etc.). Los cuatro métodos más usados son:

GET. Obtiene una representación del recurso. El cliente envía una solicitud GET para pedir la representación de un recurso, identificado por una URL.

DELETE. Destruye el recurso. El cliente envía un DELETE cuando desea que un recurso desaparezca.

POST. El método POST tiene dos trabajos. El primero es POST-to-append, en el cual, cuando se envía un POST-to-append a un recurso, se crea un nuevo recurso debajo de este. Cuando un cliente envía una petición POST-to-append, este envía una representación del recurso que quiere crear en el cuerpo de la petición.

El otro trabajo de POST es llamado *overloaded* POST. La especificación de HTTP dice que un POST puede ser usado para: *Proveer un bloque de datos, desde el resultado de enviar un formulario, hasta un proceso de manejo de datos.* El *proceso de manejo de datos* puede ser cualquier cosa. Es “legal” enviar cualquier dato como parte de la petición POST, para cualquier propósito. La definición es tan vaga que una petición POST no tiene una semántica de protocolo. En pocas palabras POST no significa “crear un nuevo recurso” significa cualquier cosa. Usar un POST para realizar un PUT, DELETE, PATCH, etc., es un overloaded POST.

PUT. Reemplaza el estado del recurso con uno dado en la representación. Una petición PUT, es una solicitud para modificar el estado de un recurso. El cliente toma la representación obtenida de GET, la modifica, y envía de regreso dentro de la petición PUT.

Existe otro método que no está definido en el estándar HTTP pero sí en el apéndice RFC 5789:

PATCH. Modifica parte de el estado del recurso basado en una representación dada. Si algún pedazo del estado del recurso no es mencionando en la representación, lo deja como está. PATCH es como PUT pero permite pequeños cambios en el estado del recurso.

Status codes

Todo mensaje de respuesta HTTP trae consigo un código de estado. El código de estado es un código de tres dígitos que le dice al cliente si la petición fue exitosa o si otras acciones son requeridas. En la Tabla 1.4.1 se muestran algunas categorías de los códigos de estado más comunes.

Tabla 1.8: Alguno códigos de estado comunes.

Categoría	Descripción
1xx: Informativo	Esta clase de códigos indican una respuesta provisional.
2xx: Éxito	Indican que la petición del cliente fu aceptada exitosamente.
3xx: Redirección	Indica que el cliente debe realizar acciones adicionales para completar la petición.
4xx: Error del cliente	Esta categoría es para casos en las que el cliente parece haberse equivocado.
5xx: Error del servidor	El servidor tomar responsabilidad del error.

Mensajes

Los mensajes HTTP son secuencias de caracteres simples. Debido a que son texto plano, y no binarios, son fáciles de leer y escribir por humanos. Se llaman mensajes de petición si son enviados desde algún cliente web a un servidor web. Los mensajes de los servidores a los clientes se conocen como mensajes de respuesta.

Un mensaje HTTP consiste de tres partes:

Línea inicial. Indica qué hacer cuando se realiza una petición o qué sucede cuando se envía una respuesta.

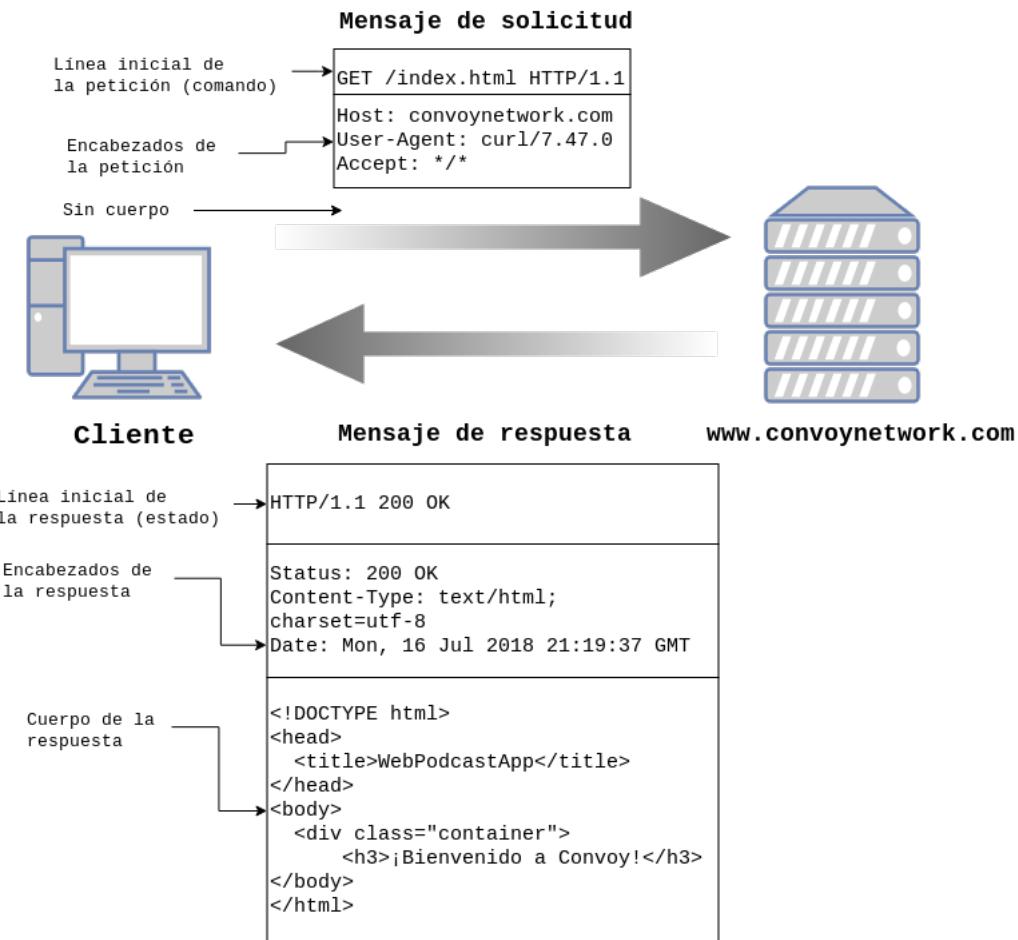


Figura 1.18: Ejemplo de la estructura de un mensaje de solicitud y de respuesta. El cliente solicita el archivo `index.html` mediante un `GET` y el servidor le regresa el archivo con un código de estado 200.

Campos del encabezado. Cero o más campos de encabezado siguen después de la línea inicial. Cada campo consiste de un nombre y un valor, separado por un punto y coma. Los encabezados terminan con un espacio en blanco.

Cuerpo. Después del espacio en blanco sigue un cuerpo del mensaje opcional que contiene cualquier tipo de datos. Los cuerpos de la petición llevan datos al servidor web; los cuerpos de la respuesta cargan datos de regreso al cliente. A diferencia de la línea inicial y los encabezados, el cuerpo puede contener datos binarios arbitrarios (imágenes, videos, audio, aplicaciones, etc.).

1.4.2. World Wide Web

La World Wide Web o Web (Red Informática Mundial en español) es un espacio de información en el cual los objetos de interés, conocidos como *recursos*, son identificados por Identificadores de Recursos Uniformes (URI por sus siglas en inglés). En diciembre de 1990, Tim Bernes-Lee comenzó este proyecto, donde inventó e implementó, entre otras cosas:

- El Identificador de Recursos Uniforme, una sintaxis que asigna a cada documento web una dirección única.
- El Protocolo de Transferencia de Hipertexto (HTTP), un lenguaje basado en mensajes que las computadoras pueden usar para comunicarse a través de Internet.
- El Lenguaje de Marcas de Hipertexto (HTML), para representar documentos informativos que contienen enlaces a documentos relacionados.
- El primer servidor web.
- El primer navegador web, llamado «Nexus».

Desde el momento que el proyecto se publicó, comenzó a crecer exponencialmente. El tráfico de la web estaba superando la capacidad de la infraestructura del Internet.

Arquitectura web

A finales de 1993, Roy Fielding, a través de un análisis, reconoció que la escalabilidad de la web estaba gobernada por un conjunto de restricciones clave. Fielding agrupó en seis categorías esas restricciones y de manera manera colectiva se refirió a ellas como el *estilo arquitectónico de la web*.

Cliente servidor

La web es un sistema basado en la dupla cliente-servidor, en la cual clientes y servidores tienen distintos papeles. Estos pueden ser implementados y desarrollados independientemente, usando cualquier lenguaje o tecnología, con tal que de que se ajusten a la *interfaz uniforme de la web*.

Interfaz uniforme

Las interacciones entre los componentes web (clientes, servidores e intermediarios), dependen de la uniformidad en sus interfaces. Si cualesquiera de estos componentes

se salen de los estándares establecidos, entonces la comunicación en la web se rompe. Los componentes web interactúan consistentemente dentro las siguientes cuatro restricciones de la interfaz uniforme.

Identificación de recursos Como se mencionó anteriormente, los elementos dentro de la web son conocidos como *recursos* y tienen un identificador único. Por ejemplo, una página web como <http://convoynetwork.com/>, identifica de manera única el recurso en la raíz del sitio web.

Manipulación de recursos a través de sus representaciones Los clientes manipulan representaciones de recursos. Un mismo recurso puede ser representado de diferentes formas a diferentes clientes. Por ejemplo, un documento puede representarse como un HTML para un navegador web, y un JSON para un programa automatizado. La idea principal es que la representación es una manera de interactuar con el recurso pero no es el recurso en sí.

Mensajes autodescriptivos El estado deseado de un recurso puede ser representado dentro del mensaje de petición del cliente. El estado actual de un recurso puede estar representado dentro del mensaje de respuesta por parte del servidor. Los mensajes autodescriptivos pueden incluir metadatos para comunicar detalles adicionales con respecto al estado del recurso, el formato de representación y tamaño, y el mensaje mismo. Un mensaje HTTP provee *encabezados* para organizar varios tipos de metadatos en campos uniformes.

Hipermedia como el motor del estado de la aplicación (HATEOAS) La representación del estado de un recurso incluye enlaces a recursos relacionados. Los enlaces son los hilos que entrelazan la web, permitiendo a los usuarios moverse a través de aplicaciones y datos de una manera directa y significativa. La presencia, o ausencia, de un enlace en una página es una parte importante del estado del recurso actual.

Sistema de capas

Las restricciones del sistema en capas permiten intermediarios de red como los proxies y puertas de acceso sean desplegadas de manera transparente entre el cliente y el servidor usando una interfaz uniforme de la web. Intermediarios basados en la red se usan para reforzar la seguridad, almacenamiento en caché de respuestas, y balanceo de carga.

Caché

El almacenamiento en caché es una de las restricciones más importantes de la arquitectura de la web. Las restricciones de la caché dan órdenes a un servidor web para declarar la *cacheabilidad* de cada datos de la respuesta. Guardar en la caché datos de la respuesta pueden ayudar a reducir la latencia percibida por el cliente, aumentar la disponibilidad general y confiabilidad de una aplicación, y controlar la carga de un servidor web. En resumen, el almacenamiento en caché reduce el *costo* general de la Web.

Stateless

La restricción stateless dicta que un servidor web no está obligado a memorizar el estado de sus aplicaciones cliente. Como resultado, cada cliente debe incluir toda la información contextual que considere sea relevante en cada interacción con el servidor web. Los servidores web solicitan a los clientes administrar la complejidad de comunicar el estado de su aplicación tal que el servidor web puede servir a un número mayor de clientes.

Código bajo demanda

Esta restricción permite a los servidores web transferir temporalmente programas ejecutables, tales como scripts o extensiones a los cliente. El código bajo demanda tiende a establecer un acoplamiento de tecnología entre servidores web y sus clientes, ya que el cliente debe tener la habilidad de entender y ejecutar el código que descarga cuando lo desea desde el servidor. Por esta razón, el código bajo demanda es la única restricción del estilo arquitectónico de la web que se considera opcional.

1.4.3. Transferencia de Estado Representacional (REST)

En el año 2000, después de que la crisis de la escalabilidad de la web se evitó, Fielding llamó y describió al estilo arquitectónico de la web en su trabajo doctoral. **Transferencia de Estado Representacional** (REST por sus siglas en inglés) fue el nombre que Fielding dio a su descripción del estilo arquitectónico de la web, que está compuesto por las restricciones mencionadas previamente.

La Transferencia de Estado Representacional es un estilo de arquitectura de software. Este estilo es una abstracción de elementos arquitectónicos dentro de un sistema de hipermedia distribuido como es la Web. REST ignora los detalles de la implementación de componentes y sintaxis de protocolos de manera que pueda enfocarse

en los papeles de los componentes, las restricciones sobre su interacción con otros componentes, y la interpretación de elementos de datos significativos. Abarca las limitaciones fundamentales sobre los componentes, conectores y datos que definen las bases de la arquitectura web y, por lo tanto, la esencia de su comportamiento como una aplicación basada en red. REST no es un estándar, sin embargo sí un conjunto de restricciones. No está atado al protocolo HTTP, pero a menudo se asocia con éste.

API REST

Un *servicio web* es un sistema de software diseñado para admitir la interacción interoperable de una máquina a otra máquina a través de una red. Programas cliente usan *interfaces de programación de aplicaciones* (API por sus siglas en inglés) para comunicarse con servicios web. Generalmente hablando, una API expone un conjunto de datos y funciones para facilitar interacciones entre programas de computadora y permitiendo que intercambien información.

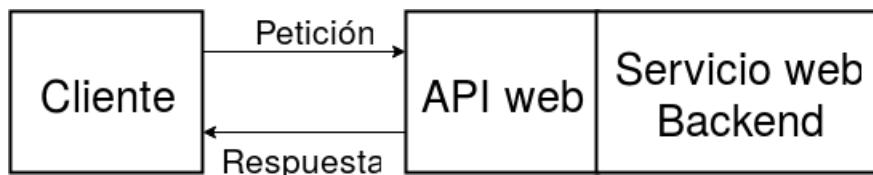


Figura 1.19: Una API web es el frente de un servicio web, escuchando y respondiendo las peticiones de los cliente directamente.

El estilo arquitectónico REST se aplica comúnmente al diseño de API para servicios web modernos. Una API web que sigue el estilo REST es una API REST. Tener una API REST hace a un servicio web RESTful. Una API REST está formada de recursos entrelazados.

Remark. *REST es una arquitectura basada en recursos. Se accede a un recurso a través de una interfaz común basada en los métodos estándar de HTTP. REST solicita a los desarrolladores usar métodos HTTP explícitamente y de una forma que sea consistente con la definición del protocolo. Cada recurso se identifica con un URL. Todos los recursos deben soportar las operaciones HTTP más comunes, además REST permite que ese recurso tenga diferentes representaciones, por ejemplo, texto, xml, json, etc. El cliente REST puede solicitar una representación específica por medio del protocolo HTTP. La Tabla 1.4.3 describe los elementos usados en REST.*

Tabla 1.9: Elementos de REST

Elemento	Descripción
Recurso	Objetivo conceptual de una referencia de hipertexto. Por ejemplo: podcast.
Identificador de recurso	Un URL que identifica un recurso en específico. Por ejemplo: http://convoynetwork.com/podcast/123
Metadatos del recurso	Información que describe al recurso. Por ejemplo: autor, etiqueta, etc.
Representación	El contenido del recurso. Por ejemplo: un JSON, un HTML o una imagen JPEG.
Metadatos de la representación	Información que describe como procesar la representación. Por ejemplo: tipo de medio, fecha, etc.
Datos de control	Información que describe cómo optimizar el procesamiento de respuesta. Por ejemplo: if-modified-since, cache-control-expiry.

1.5. Servicios de BaaS y Firebase

1.5.1. Backend as a Service

Remark. *Backend as a service o Mobile Backend as a Service, es un modelo para proporcionar a los desarrolladores web y de aplicaciones móviles una forma de vincular estas aplicaciones al almacenamiento en la nube, servicios analíticos y/o otras características tales como la gestión de usuarios, la posibilidad de enviar notificaciones push y la integración con servicios de redes sociales. Estos servicios son brindados a través de kits de desarrollo de software (SDK) o interfaces de programación de aplicaciones (API).*

Algunos proveedores de este tipo de cómputo en la nube son Microsoft Azure Mobile Services, Oracle Cloud (Mobile Service) y **Firebase**, que es la plataforma que se usa en este proyecto y por tanto se describe a detalle a continuación.

1.5.2. Firebase

Firebase es una plataforma de desarrollo para aplicaciones móviles y web. Tiene varias tecnologías para mejorar la experiencia de desarrollo de una aplicación, como la autenticación, base de datos en tiempo real, almacenamiento en la nube,

alojamiento de un sitio estático y funciones en la nube para ejecutar código de backend.

A pesar de ser una plataforma enfocada principalmente para ser un servicio de backend para aplicaciones móviles y web, algunos de sus servicios cuentan con una API REST lo que permite acceder a estos a través de entornos con recursos restringidos.

De las tecnologías ofrecidas por Firebase, se usan:

- Realtime database
- Authentication
- Hosting

Firebase Realtime Database

Firebase Realtime Database es una base de datos alojada en la nube. Los datos se almacenan en formato JSON y se sincronizan en tiempo real con cada cliente conectado. Los clientes comparten una instancia de Realtime Database y reciben actualizaciones automáticamente con los datos más recientes.

Entre las funciones más importantes de este servicio están; la **sincronización de datos en tiempo real**, cada vez que los datos cambian, los dispositivos conectados reciben esa actualización en milisegundos. Trabajo **sin conexión**, ya que el SDK hace que los datos persistan en el disco, y cuando la conexión se restablece, el dispositivo cliente recibe los cambios faltantes y los sincroniza con el estado actual del servidor. **Accesible desde dispositivos cliente**, no se necesita un servidor de aplicaciones; las opciones de seguridad y validación se definen a través de reglas de seguridad de Firebase Realtime Database. Y finalmente el **escalamiento en varias bases de datos**, que permite dividir la información en diversas instancias de bases de datos dentro del mismo proyecto de Firebase.

La ruta de implementación de la Realtime Database que está definida en la documentación de Firebase, es la siguiente:

1. Integrar los SDK de Firebase Realtime Database.
2. Crear referencias de Realtime Database.
3. Configurar datos y escuchar para detectar cambios.
4. Habilitar la persistencia sin conexión.
5. Proteger los datos con las reglas de seguridad de Firebase Realtime Database.

Firebase Authentication

La mayoría de las aplicaciones necesitan identificar a los usuarios. Conocer la identidad de un usuario permite que una aplicación guarde sus datos en la nube de forma segura y proporcione la misma experiencia personalizada en todos los dispositivos del usuario.

Firebase Authentication proporciona servicios de backend, SDK fáciles de usar y bibliotecas de IU ya elaboradas para autenticar a los usuarios en tu aplicación. Admite la autenticación mediante contraseñas, números de teléfono, proveedores de identidad federados populares, como Google, Facebook y Twitter, y mucho más.

Firebase Authentication se integra estrechamente con otros servicios de Firebase y aprovecha los estándares de la industria como OAuth 2.0 y OpenID Connect, por lo que se puede integrar fácilmente con tu backend personalizado.

FirebaseUI Auth

Puedes permitir que los usuarios accedan a tu aplicación de Firebase con FirebaseUI como solución de autenticación directa o mediante el SDK de Firebase Authentication para integrar de forma manual uno o más métodos de acceso en tu aplicación.

FirebaseUI proporciona una solución de autenticación directa que controla los flujos de IU para los usuarios que acceden con direcciones de correo electrónico y contraseñas, números de teléfono y con proveedores de identidad federada populares, que incluyen el Acceso con Google y el Acceso con Facebook.

El componente de FirebaseUI Auth implementa recomendaciones para la autenticación en sitios web y dispositivos móviles, lo que puede maximizar la conversión de acceso y registro de tu aplicación. También maneja casos extremos, como recuperación y vinculación de cuentas, que pueden tener repercusiones en la seguridad y ser propensos a generar errores cuando se tratan de manejar correctamente.

Firebase Hosting

Firebase Hosting proporciona hosting estático, rápido y seguro para tu aplicación web.

Firebase Hosting es un servicio de hosting de contenido web con nivel de producción orientado a programadores. Con Hosting, puedes implementar aplicaciones web y contenido estático en una red de distribución de contenido global (CDN) con un solo comando, en forma rápida y sencilla.

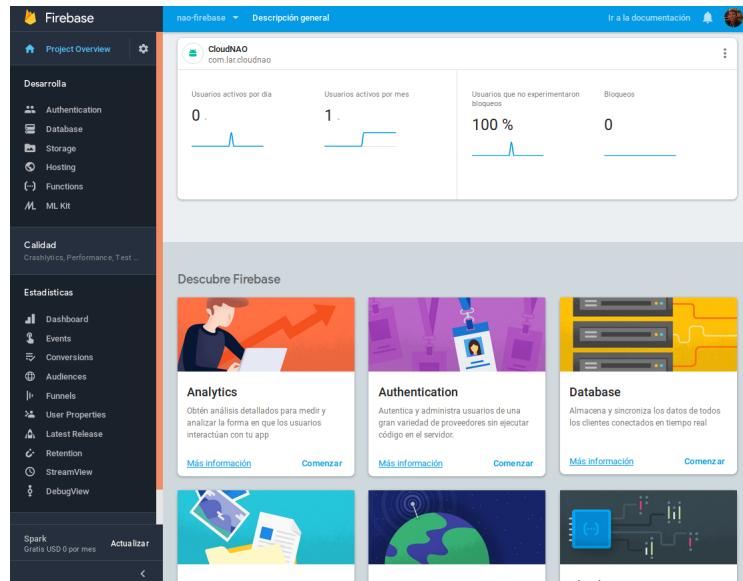
La ruta de implementación para utilizar el servicio de alojamiento en una aplicación web es la siguiente:

1. Instala Firebase CLI. Con Firebase CLI, es fácil configurar un nuevo proyecto de Hosting, administrar un servidor de desarrollo local y también implementar contenido
2. Configura un directorio de proyectos. Agrega archivos para tu aplicación web y agrega tus activos estáticos a tu carpeta local de Hosting. A continuación, puedes ejecutar `firebase serve` para tu sitio en forma local.
3. Implementa tu sitio. Cuando estés satisfecho con la configuración, ejecuta `firebase deploy` para subir la última instantánea a nuestros servidores.

Firebase Console

La consola de Firebase es una interfaz gráfica de usuario, GUI por sus siglas en inglés, para que a través de un navegador se administren los servicios provistos por Firebase. Esta consola permite a los desarrolladores crear proyectos, añadir servicios a esos proyectos, configurarlos, administrar las herramientas, visualizar estadísticas de las aplicaciones, entre muchas cosas más.

Figura 1.20: Una captura de un proyecto visto desde la consola de Firebase.



1.6. Interfaces de Programación de Aplicaciones de Transferencia de Estado Representacional

Para entender mejor el concepto de una API REST conviene poner como ejemplos los servicios web RESTful que se utilizan en este proyecto. Se accede a esos servicios a través de sus API REST y se enlistan a continuación:

- Google Cloud Vision
- Kairos
- Wit.ai
- Firebase Realtime Database

1.6.1. API de Google Cloud Vision

La API Vision de Google Cloud permite que los desarrolladores comprendan el contenido de una imagen mediante el encapsulado de potentes modelos de aprendizaje automático en una API REST fácil de usar. La API clasifica imágenes rápidamente en miles de categorías (por ejemplo, «barco de vela», «león» o «torre Eiffel»), detecta objetos y caras individuales dentro de las imágenes, además de buscar y leer palabras impresas en ellas.

El recurso que se solicita para el procesamiento de imágenes es `images` y el URL que lo identifica es `https://vision.googleapis.com/v1/images:annotate`. La API únicamente tiene definida la operación con el método HTTP POST, por lo que hacer un POST a la URL del recurso ejecuta la detección de algunas características en una o varias imágenes.

El cuerpo del mensaje del petición HTTP es un JSON con la siguiente estructura (por simplicidad se omiten varios objetos del JSON que no se utilizaron en el proyecto):

```
{  
  "requests" : [  
    {  
      "image" : {  
        "content" : string,  
        "source" : {  
          "gcsImageUri" : string,  
          "imageUri" : string  
        }  
      }  
    }  
  ]  
}
```

```

    }
    "features" : [
      {
        "type" : string,
        "maxResults" : number
      }
    ]
  ]
}

```

`requests` es una lista de objetos, donde cada objeto representa una imagen y las caracterísiticas que se piden detectar. `image` es la representación de la imagen, que puede estar en tres formas distintas: codificada en base 64, guardada en el servicio de almacenamiento en la nuble de Google Cloud Storage o a través de un enlace en algún servidor. Para el primer caso a `content` se le asigna la image codificada; en cambio, si está en un contenedor de Google Cloud Storage entonces se pasa a `gcsImageUri` el valor de la URL de la imagen. Si se encuentra en algún otro servicio de almacenamiento simplemente el valor de `imageUri` es el URL de la imagen.

`features` es un arreglo de las caracterísiticas que se pretenden reconocer en la imagen. Por cada objeto se tiene `type` que es el tipo de caracterísitica y `maxResults` que es el número máximo de ese tipo. `type` puede tomar los siguientes valores:

- `TYPE_UNSPECIFIED`: No se especifica la característica.
- `FACE_DETECTION`: Ejecuta la detección de rostros.
- `LANDMARK_DETECTION`: Ejecuta la detección de puntos de referencia.
- `LOGO_DETECTION`: Ejecuta la detección de logotipos.
- `LABEL_DETECTION`: Ejecuta la detección de etiquetas.
- `TEXT_DETECTION`: Ejecuta el reconocimiento óptico de caracteres (OCR).
- `DOCUMENT_TEXT_DETECTION`: Ejecuta el OCR sobre grandes documentos de texto.
- `SAFE_SEARCH_DETECTION`: Ejecuta la búsqueda segura para detectar contenido potencialmente inseguro o no deseable.
- `IMAGE_PROPERTIES`: Calcula un conjunto de propiedades en una imagen, como los colores dominantes.
- `CROP_HINTS`: Ejecuta la sugerencia de recortes de una imagen.
- `WEB_DETECTION`: Ejecuta la detección web.

De todas estas posibles características, las que se utilizan dentro de la arquitectura CloudNAO son `TEXT_DETECTION` y `LABEL_DETECTION`.

El cuerpo del mensaje de respuesta, si todo salió bien (se obtiene una código de estado 200), contiene información con la siguiente estructura:

```
{
  "responses": [
    {
      "labelAnnotations": [
        {
          "mid": string,
          "description": string,
          "score": number
        }
      ],
      "textAnnotations": [
        {
          "mid": string,
          "locale": string,
          "score": number,
          "boundingPoly": {
            "vertices": [
              {
                "x": number,
                "y": number
              }
            ]
          }
        }
      ]
    }
  ]
}
```

La respuesta obtenida depende de que características se solicitaron, para el JSON anterior suponemos que se pidió la detección de categorías y el reconocimiento de texto. `mid` es un identificador, `description` es la descripción textual de la entidad; por ejemplo, para el caso de `labelAnnotations`, `description` puede tener un valor asociado igual a «psychedelic art», en cambio para `textAnnotations`, se espera tenga una cadena del texto encontrado como «lateralus».

El `score` es la precisión de la detección de la entidad en la imagen, toma valores en el intervalo [0, 1]. `boundingPoly` contiene las coordenadas del polígono que encierra la entidad encontrada.

1.6.2. API de Kairos

Kairos permite a los desarrolladores integrar un análisis de rostros preciso y rápido en cualquier aplicación o servicio. Cuenta con un API REST para ejecutar tareas en la nube como la detección e identificación de rostros, emociones, raza, edad, entre otras. También cuenta con un SDK, más limitado pero que puede ejecutarse offline y directamente en dispositivos móviles. Por ahora sólo nos interesa la API REST, por lo que describiremos cómo funciona, es decir, cómo se debe hacer una petición y cuál es la respuesta que envía el servicio.

Para realizar el reconocimiento de caras, la API tiene como URL base <https://api.kairos.com> y cuenta con diversos recursos para cada tipo de reconocimiento. Únicamente nos interesan dos recursos, `enroll` y `recognize`, que son relativos al URL base. El primero toma una fotografía, encuentra rostros en esta, y los guarda en una galería, cada rostro necesita un identificador de la persona. Las galerías sirven para que posteriormente se reconozca a un sujeto por medio de su fotografía.

El otro recurso que nos interesa es `recognize`, que toma una fotografía encuentra caras e intenta emparejarlas con las que sean guardado previamente en una galería.

Para hacer ambas peticiones se utiliza el método POST. Al hacer un POST a `/enroll` se debe enviar un JSON con la siguiente estructura en el cuerpo del mensaje.

```
{  
  "image": string,  
  "subject_id": string,  
  "gallery_name": string  
}
```

`image` es un URL de acceso público o la codificación en base 64, `subject_id` es un identificador de una cara, es único y es definido por el desarrollador, `gallery_name` definida también por el desarrollador sirve para identificar la galería.

La respuesta de un POST a `/enroll`, si todo fue exitoso es un JSON con la siguiente estructura.

```
{  
  "face_id": string,  
  "images": [  
    {  
      "attributes": {  
        "lips": string,  
        "asian": number,  
        "gender": {  
          "type": string  
        }  
      }  
    }  
  ]  
}
```

```

},
  "age": number,
  "hispanic": number,
  "other": number,
  "black": number,
  "white": number,
  "glasses": string
},
  "transaction": {
    "status": string,
    "topLeftX": number,
    "topLeftY": number,
    "gallery_name": string,
    "timestamp": string,
    "height": number,
    "quality": number,
    "confidence": number,
    "subject_id": string,
    "width": number,
    "face_id": number
  }
}
]
}

```

Como se puede ver, los elementos del JSON son autodescriptivos. Para el caso del POST a /recognize, el cuerpo de la petición lleva un JSON como el siguiente:

```
{
  "image": string,
  "gallery_name": string
}
```

La respuesta enviada por Kairos, si se todo fue exitoso, es un JSON con campos autodescriptivos como los de la respuesta del recurso previo. Este JSON tiene se muestra a continuación.

```
{
  "images": [
    {
      "transaction": {
        "status": string,
        "width": number,
        "topLeftX": number,

```

```

        "topLeftY": number,
        "gallery_name": string,
        "face_id": number,
        "confidence": number,
        "subject_id": string,
        "height": number,
        "quality": number
    },
    "candidates": [
        {
            "subject_id": string,
            "confidence": number,
            "enrollment_timestamp": string
        },
        {
            "subject_id": string,
            "confidence": numref,
            "enrollment_timestamp": string
        }
    ]
}

```

1.6.3. API de Wit.ai

Wit.ai es una API para procesamiento de lenguaje natural capaz de convertir oraciones en datos estructurados. Esto quiere decir que puedes crear bots que interactúen con humanos en plataformas de mensajería. Con Wit.ai los desarrolladores pueden construir aplicaciones a la que se les pueda hablar o escribir. Los bots aprenden conforme se les hable, se vuelven más inteligentes en cada interacción.

Wit.ai, como muchos servicios, cuenta con diferentes SDK para lenguajes como JavaScript, Python y Ruby. También tiene una API REST para extraer el significado de una sentencia o de un discurso de audio. Por ahora nos enfocaremos al segundo caso, cómo solicitar y cuál es la respuesta al enviar audio para su procesamiento.

Wit trabaja con *intents* y *entities*. Las intents se usan para entender el significado en general de una oración. Por ejemplo la intent de la oración «¿A qué hora abren la biblioteca central de CU?» puede ser las horas de apertura de un lugar. Las entities proveen más información para contestar correctamente. Siguiente el ejemplo, el usuario no está preguntando a qué hora abren todas las bibliotecas, sino en específico

la biblioteca central de CU, por lo que una entity sería el lugar del que se deseaba obtener la hora de apertura.

La API cuenta con el URL base `https://api.wit.ai/` y el recurso que solicitamos es `/speech`. A través de un POST a `https://api.wit.ai/speech` podemos obtener el significado de un archivo de audio.

A diferencia de las otras API donde no necesariamente tenemos que especificar un encabezado en el mensaje de petición HTTP, aquí es necesario especificar el formato del archivo de audio. Se debe pasar como valor al campo `Content-type` cualquiera de los siguientes:

- `audio/wav`: si envias un archivo wav.
- `audio/mpeg3`: para un archivo mp3.
- `audio/ulaw`: para archivo G.11 u-law.

Wit.ai únicamente procesa audio en formato monoaural no estéreo.

En el cuerpo del mensaje va la información binaria, es decir, el archivo de audio. La respuesta del servidor es un JSON con la siguiente estructura:

```
{
  "msg_id": string,
  "_text": string,
  "entities": {
    string : [ {
      "value": string,
      "confidence": number
    } ],
    "intent": {
      "value": string,
      "confidence": number
    }
  }
}
```

`msg_id` es el id de la respuesta. `_text` es la transcripción del discurso de audio en texto. `entities` es un objeto con las entities e intents que se encontraron. Si es una intent el objeto tiene como llave la cadena `"intent"` y como valores los campos `value` y `confidence`, donde el primero es el valor con el que se definió la intent y el segundo es la probabilidad de que sea tal intent. Si es una entity la llave del objeto es el nombre de la entity y como valor tiene un arreglo de las diferentes apariciones que tiene esa entity en el mensaje. Cada entity tiene los campos de `value` y `confidence`.

1.6.4. API REST de Firebase Database

La base de datos en tiempo real de Firebase cuenta con una API REST para realizar operaciones para leer, escribir, modificar o eliminar datos.

Se puede utilizar cualquier URL de la base de datos de Firebase como un endpoint REST. Todo lo que se necesita es añadir la extensión `.json` al final del URL y enviar una petición desde cualquier cliente HTTP.

Para poder utilizar la API REST, el mapeo de los métodos HTTP y las operaciones sobre Firebase Realtime Database es el siguiente.

- **GET**, para leer información previamente almacenada en la base de datos.
- **PUT**, para escribir datos sobre una ubicación de la base de datos. Si existen datos los sobreescribe. Equivalente a `set()` de los SDK.
- **POST**, para añadir información en la base de datos. Se asigna un identificador único generado por Firebase. Es equivalente al `push()` de los SDK.
- **PATCH**, para actualizar datos en una ubicación sin sobreescribir. Equivalente a `update()` de los SDK.
- **DELETE**, elimina los datos en la ubicación dada.

La lista anterior muestra los métodos de lectura y escritura sobre la base de datos, sin embargo, la API REST soportan el protocolo **Server-sent events (SSE)**, que define una API para recibir notificaciones push desde un servidor a través de una conexión HTTP.

Para recibir las actualizaciones sobre una ubicación en la base de datos se necesitan hacer tres cosas:

- Configurar el header `Accept` igual a `text/event-stream`. Del lado del cliente.
- Respetar los redireccionamientos HTTP.
- Si la ubicación de la base de datos requiere permisos, incluir el parámetro `auth`.

Capítulo 2

La arquitectura CloudNAO

2.1. Descripción de la arquitectura CloudNAO

CloudNAO integra servicios web de terceros, servicios web desarrollados por el LAR, una aplicación móvil para dispositivos con el sistema operativo Android, la plataforma como servicio Firebase, una aplicación web y al robot humanoide NAO. Todos estos componentes se comunican a través de internet. Cada componente tiene subcomponentes que permiten la comunicación y el procesamiento de diversas tareas. En la parte del backend están los servicios web de terceros, los desarrollados por el LAR y Firebase. Los elementos restantes corresponden entonces al frontend.

Los servicios web de terceros utilizados son brindados por Google Cloud, Kairos y Wit.ai. Se integran a algunos elementos de la arquitectura a través de sus API REST.

Los servicios provistos por el LAR, son ejemplos de algunas aplicaciones sobre el robot que le permitan realizar tareas complejas, como el procesamiento de imágenes para detección de objetos o clasificación de escenarios.

Los servicios de terceros junto con los del LAR se integran dentro de un servidor al que llamaremos *servidor LAR*. Éste cuenta con una API REST, que tiene como clientes al robot y a la aplicación móvil.

Las funcionalidades de Firebase utilizadas son: la base de datos en tiempo real, para envío de información entre el robot y las aplicaciones móvil y web, la autenticación, para el acceso a la aplicación móvil y web, y hosting para la aplicación web.

La aplicación móvil es una herramienta frontend para que los usuarios interactúen con el robot sin necesidad de instalar o ejecutar un programa de manera local en éste. Hace peticiones a la API REST del servidor del LAR y se comunica con

Firebase. Usando el SDK de Java de NAOqi la aplicación recibe datos del robot y ejecuta comandos de manera remota.

De manera similar, la aplicación web, es una parte del frontend para enlazar al usuario con el robot, sin embargo, esta interacción tiene como intermediario a Firebase, por lo que la aplicación utiliza su API web. En este caso el robot sí necesita comunicarse con Firebase a través de un programa corriendo local o remotamente.

El robot es quien se conecta con el mayor número de componentes, usando las API web de los servicios o plataformas, y con el framework de NAOqi.

La Figura 2.1 es un diagrama con los componentes de la arquitectura y de las interfaces que brindan o necesita cada uno de los elementos.

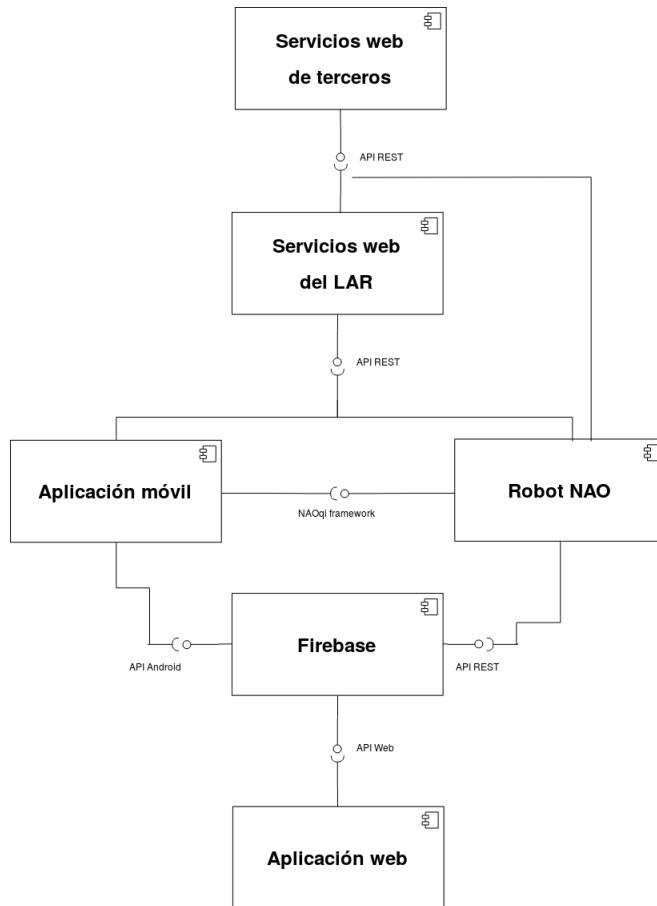


Figura 2.1: Diagrama de los componentes de la arquitectura.

En las siguientes secciones se describen con mayor detalle cada uno de los componentes y subcomponentes que forman esta arquitectura.

2.1.1. Servidor LAR

El servidor LAR es cliente de los servicios web de terceros y a la vez ofrece los mantenidos por el LAR. El resultado de la unión cliente/servidor se entrega por medio de una API REST. Todos los servicios brindados tienen en común que están basados en modelos de aprendizaje automático, específicamente de aprendizaje profundo. Estos dan solución a problemas de visión computacional como la detección de rostros o el reconocimiento óptico de caracteres, el reconocimiento de voz, el procesamiento de lenguaje natural y la traducción automática neuronal. A continuación se enlistan los servicios de terceros que el servidor consume.

- API Vision de Google Cloud, permite comprender el contenido de una imagen. Son dos las funcionalidades ocupadas; el etiquetado de imágenes en miles de categorías y el reconocimiento óptico de caracteres (OCR).
- API Translation de Google Cloud, se encarga de traducir una cadena arbitraria en cualquier idioma admitido.
- Kairos, es una API de reconocimiento facial. Esta cuenta con varios métodos, de los cuales se utilizaron *enroll* y *recognize*. El primero para añadir a un nuevo rostro a una base de datos junto con un identificador, el segundo para encontrar a un usuario cuya cara ha sido almacenada.
- Wit.ai, una API para procesamiento de lenguaje natural capaz de convertir oraciones en información estructurada.

El servidor mantiene la ejecución de un contenedor de Docker en el que se encuentra corriendo una aplicación web desarrollada en Python. Dicha aplicación es la API REST que se comunica con el robot y la aplicación móvil.

La Figura 2.2 muestra de manera general la estructura del servidor.

2.1.2. API REST de CloudNAO

Esta interfaz es el producto de la integración de modelos de aprendizaje automático enfocados a casos de uso relacionados con la robótica. Es uno de los componentes más importantes en toda la arquitectura, es el que une los servicios web de terceros, los módulos mantenidos por el LAR y los entrega en una sola API. Es un cliente y servidor a la vez.

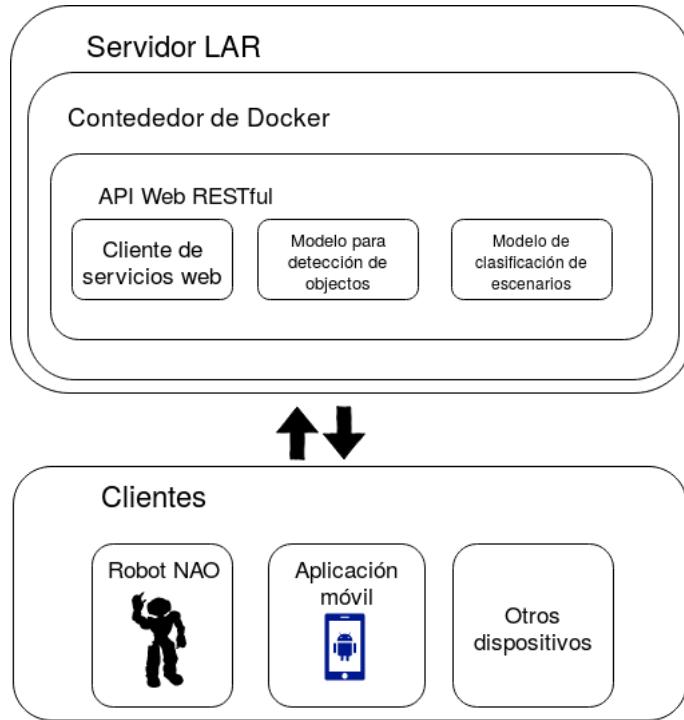


Figura 2.2: Diagrama del servidor

Descripción de la API

La API REST de CloudNAO permite integrar dentro de una aplicación un conjunto de herramientas para el análisis de imágenes. Todas éstas basadas en modelos de aprendizaje automático. El URL base al que todos los recursos son relativos es <http://132.248.180.17/>.

La API tiene tres recursos:

- **/register**: para que un nuevo usuario se registre y pueda ocupar los otros recursos. Envía una dirección de correo y una contraseña.
- **/refreshtoken**: para que el usuario pueda obtener un nuevo token de acceso al enviar los datos con los que se registró.
- **/vision**: el recurso más importante. Se le solicita que haga procesamiento de imágenes.

Vision

Este recurso es el encargado de la detección de características en una imagen. Estas características son: la **detección de objetos**, el **reconocimiento de rostros**, de personas previamente guardadas o de nuevos sujetos para su almacenamiento, la **clasificación en cuatro escenarios**, lugares dentro del laboratorio de algoritmos para la robótica, la **detección de etiquetas o categorías** y la **traducción de texto encontrado en una imagen**.

En este recurso la única forma de persistencia de datos es al guardar el rostro de una nueva persona para su posterior reconocimiento.

POST /vision

Se envía una imagen ya sea codificada en base 64 o mediante su URL y las características que se desean obtener. Las características disponibles son las siguientes:

- **FACE_ENROLL:** Detecta un rostro en la imagen y con un identificador enviado en el cuerpo de la petición se almacena en una galería de Kairos. Se emplea /enroll de Kairos.
- **FACE_RECOGNITION:** Encuentra rostros dentro de una imagen y los relaciona con los rostros similares previamente guardados en una galería de Kairos. Usa /recognize de la API de Kairos.
- **OBJECT_DETECTION:** Busca objetos en una la imagen, usa la API de Tensorflow object detection. Regresa las coordenadas de un cuadro delimitador para cada objeto detectado.
- **LABELS_DETECTION:** Clasifica una imagen en distintas categorías. Se vale de la API de Google Cloud Vision
- **OCR_TRANSLATION:** Lleva a cabo el reconocimiento de texto en una imagen, para posteriormente traducirlo. Utiliza la API de Google Cloud Vision para el reconocimiento de caracteres y la de Google Cloud Translation para la segunda parte del proceso.
- **CLASSIFY_INDOOR_SCENES:** Clasifica una imagen en cuatro categorías. Cada categoría es un lugar dentro del área del LAR.

Solicitud

Headers. En los encabezados de la petición debe de ir el tipo de contenido que se envía y un token de acceso único para cada usuario registrado. Esto último

simplemente es para evitar que cualquiera pueda hacer peticiones a la API.

```
Content-Type: application/json  
Authorization: ACCESS_TOKEN
```

Cuerpo. En el cuerpo del mensaje se envía un JSON con la siguiente estructura. En la tabla 2.1 se describen los atributos del JSON.

```
{  
    "imageContent": "Hello, world!",  
    "imageSource": "Hello, world!",  
    "features": [  
        {  
            "type": "Hello, world!",  
            "subjectID": "Hello, world!"  
        }  
    ]  
}
```

Tabla 2.1: Descripción de los elementos del JSON del cuerpo de la solicitud.

Propiedades	
imageContent	
Tipo de dato	string
Descripción	Imagen codificada en base 64.
imageSource	
Tipo de dato	string
Descripción	URL pública de la imagen.
features	
Tipo de dato	array
Descripción	Una arreglo de las características que se desean detectar en la imagen. Se debe solicitar al menos una de las seis disponibles. Por ejemplo FACE_ENROLL, FACE_RECOGNITION, CLASSIFY_INDOOR_SCENES, etc.
Campos obligatorios	
imageContent	
imageSource	
features	

Respuesta

Cuepo. En el cuerpo del mensaje de respuesta si se obtiene un código 200 se obtiene un JSON con las características encontradas. En éste también se incluyen mensajes de error que no pertenecen al estándar del protocolo HTTP. En la tabla 2.2 se describen los atributos del JSON.

```
{  
  "features": [  
    "faceRecognition": [  
      {  
        "topLeftX": 1,  
        "topLeftY": 1,  
        "width": 1,  
        "height": 1,  
        "subjectId": "Hello, world!",  
        "confidence": 1  
      }  
    ],  
    "objectDetection": [  
      {  
        "category": "Hello, world!",  
        "confidence": 1,  
        "topLeftX": 1,  
        "topLeftY": 1,  
        "width": 1,  
        "height": 1  
      }  
    ],  
    "labelsDetection": [  
      {  
        "name": "Hello, world!",  
        "confidence": 1  
      }  
    ],  
    "ocrTranslation": {  
      "sourceText": "Hello, world!",  
      "targetText": "Hello, world!",  
      "sourceLanguage": "Hello, world!"  
    },  
    "faceEnroll": {  
      "topLeftX": 1,  
      "topLeftY": 1,  
      "width": 1,  
      "height": 1,  
      "confidence": 1  
    },
```

```

        "gender": "Hello, world!"
    }
    "indoorScenesClassify" : {
        "indoorScene": "Hello, world!"
    }
},
}

```

Tabla 2.2: Atributos que componen el JSON del cuerpo de la respuesta.

features	
Tipo de dato	objeto
Propiedades	
faceRecognition	
Tipo de dato	array
Descripción	Un arreglo de objetos que contienen a los rostros reconocidos.
objectDetection	
Tipo de dato	array
Descripción	Contiene un arreglo con todos los objetos detectados.
labelsDetection	
Tipo de dato	array
Descripción	Contiene un arreglo con las etiquetas de la imagen.
ocrTranslation	
Tipo de dato	objeto
Propiedades	
sourceText	
Tipo de dato	string
Descripción	El texto en formato UTF-8.
targetText	
Tipo de dato	string
Descripción	El texto traducido.
sourceLanguage	
Tipo de dato	string
Descripción	El idioma original.
Descripción	Un objeto con el texto encontrado en la imagen.
faceEnroll	
Tipo de dato	objeto
Propiedades	
topLeftX	
Tipo de dato	number

Descripción	Coordenada sobre el eje x.
topLeftY	
Tipo de dato	number
Descripción	Coordenada
width	
Tipo de dato	number
Descripción	Ancho del recuadro que delimita la imagen.
height	
Tipo de dato	number
Descripción	Altura del recuadro que delimita la imagen.
confidence	
Tipo de dato	number
Descripción	Valor de 0-1 que representa una probabilidad.
gender	
Tipo de dato	string
Descripción	Sexo de la persona con ese rostro (M o F)
Descripción	Características del rostro detectado.
indoorScenesClassify	
Tipo de dato	objeto
Propiedades	
indoorScene	
Tipo de dato	string
Descripción	La escena detectada, puede ser cualquiera de las cuatro posibles <i>exit</i> , <i>soccer_court</i> , <i>desks</i> , <i>office</i>
Descripción	Escenario reconocido.
Descripción	Lista con las respuestas de acuerdo a las características que se solicitaron.

Ejemplo de una petición a la API con cURL. cURL es una herramienta en la línea de comandos para transferir datos usando diferentes protocolos. Por facilidad, se muestra el uso de la API a través de esta herramienta. Para solicitar el recurso para procesamiento de imágenes, **vision** se necesita un token de acceso. El token se obtiene creando un usuario haciendo una petición al recurso **register**. El código para solicitar este servicio con cURL es el siguiente:

```
curl -X POST \
http://132.248.180.17/register \
-H 'content-type: application/json' \
-d '{
  "username" : "mock_user@gmailcom",
```

```
    "password" : "p45sw0rd"
}'
```

La respuesta de la API es un JSON como el que sigue:

```
{
  "message": "User created successfully.",
  "token": "0FdT6v23Tnw95S8BX2yNR80s8RB6L4HAnvhTTrfjmxB5UyMaef"
}'
```

Con el token se pueden hacer las peticiones que se deseen al recurso **visión**. Supongamos que queremos analizar una imagen para encontrar rostros, objetos, etiquetas, traducción de texto y clasificación de escenarios. El código para generar la petición es el siguiente:

```
curl -X POST \
http://132.248.180.17/vision \
-H 'authorization: 0FdT6v23Tnw95S8BX2yNR80s8RB6L4HAnvhTTrfjmxB5UyMaef' \
-H 'content-type: application/json' \
-d '{
  "imageSource" : "https://www.robotshop.com/blog/en/files/NAO-Hanover.jpg",
  "features" : [
    {
      "type": "FACE_RECOGNITION"
    },
    {
      "type" : "OBJECT_DETECTION"
    },
    {
      "type" : "LABELS_DETECTION"
    },
    {
      "type" : "OCR_TRANSLATION"
    },
    {
      "type" : "CLASSIFY_INDOOR_SCENES"
    }
  ]
}'
```

La respuesta de la API de CloudNAO envía como respuesta el subsecuente JSON:

```
{
  "features": {
```

```
"labelsDetection": [
    {
        "confidence": 0.9718034,
        "name": "robot"
    },
    {
        "confidence": 0.94625956,
        "name": "sport venue"
    },
    {
        "confidence": 0.9407438,
        "name": "technology"
    },
    {
        "confidence": 0.8791277,
        "name": "machine"
    },
    {
        "confidence": 0.76215124,
        "name": "arena football"
    },
],
"objectDetection": [
    {
        "confidence": 0.9916030764579773,
        "category": "person",
        "topLeftY": 42,
        "height": 158,
        "topLeftX": 430,
        "width": 50
    },
    {
        "confidence": 0.9870478510856628,
        "category": "person",
        "topLeftY": 79,
        "height": 115,
        "topLeftX": 253,
        "width": 49
    },
    {
        "confidence": 0.9252263307571411,
        "category": "person",
        "topLeftY": 11,
        "height": 148,
```

```

        "topLeftX": 2,
        "width": 53
    },
    {
        "confidence": 0.9083054661750793,
        "category": "sports ball",
        "topLeftY": 248,
        "height": 50,
        "topLeftX": 219,
        "width": 49
    },
],
"indoorScenesClassify": {
    "indoor_scene": "soccer_court"
},
},
"errors": {
    "faceRecognition": {
        "message": "invalid url was sent"
    },
    "ocrTranslation": {
        "message": "Text not found"
    }
}
}

```

Guía para desarrolladores

Esta API puede funcionar de manera individual, no es necesario contar con todos los componentes de la arquitectura. Puede integrarse en algún otro sistema, donde se requieran los mismos servicios web.

Está desarrollada en el lenguaje de programación Python, utilizando el framework Flask. La siguiente lista muestra las dependencias más importantes:

- **Flask**, un microframework para desarrollar aplicaciones web, y extensiones de éste como:
 - **Flask-RESTful**, para contruir rápidamente una API REST.
 - **Flask-SQLAlchemy**, Es una biblioteca que implementa la técnica ORM (Mapeador Relacional de Objetos), la cual permite consultar y manipular datos de una base de datos usando el paradigma orientado a objetos.

- **Flask-Script**, una interfaz la línea de comandos.
- **Tensorflow**, una biblioteca de código abierto para cómputo numérico usando gráficas de flujos de datos.
- **numpy**, una biblioteca para cómputo científico.
- **requests**, una biblioteca para las peticiones HTTP a servicios web.

Para comprender mejor este elemento, en las siguientes subsecciones se describen cada una de las herramientas utilizadas, primero el framework Flask, junto con sus extensiones y las otras bibliotecas mencionadas. Después explico su integración en una sola aplicación, y finalmente las opciones de configuración e instalación, así como su despliegue en un servidor.

Flask

Flask es un micro framework para desarrollo web. Fue diseñado para ser un framework extensible, sólo provee un núcleo sólido con servicios básicos, mientras que las extensiones brindan el resto. Esto quiere decir que Flask, te da la posibilidad de como desarrollador contar sólo con las dependencias que necesitas.

Tiene dos dependencias principales, **Werkzeug** y **Jinja2**. El enrutamiento, debugging y WSGI tienen procedencia de Werkzeug. Jinga2 ofrece el soporte para el manejo de plantillas. Para nuestra aplicación solamente hacemos uso de la primera.

WSGI es la Web Server Gateway Interface. Es un protocolo que describe cómo un servicio web se comunica con aplicaciones web, y cómo las aplicaciones web se pueden encadenar para procesar una petición.

Para la instalación de Flask así como de cualquier paquete de Python, se recomienda crear un ambiente virtual con la herramienta **virtualenv**. Un ambiente virtual es una copia privada del intérprete de Python sobre la cual es posible instalar paquetes de manera privada, sin afectar al intérprete global de Python. Permite crear un ambiente de trabajo aislado para cada proyecto, así cada aplicación tiene acceso solo a los paquetes que utiliza.

Con esto, todo lo descrito a continuación que se refiera a la instalación de paquetes o ejecución de programas se hace dentro de un ambiente virtual.

Inicialización. Todas las aplicaciones de Flask deben crear una *instancia de la aplicación*. Este objeto maneja todas las peticiones que recibe el servidor web, usando la especificación WSGI. La instancia de la aplicación es un objeto de la clase Flask, se crea de la siguiente manera:

```
from flask import Flask
app = Flask(__name__)
```

El único argumento requerido para el constructor de la clase Flask es el nombre del módulo o paquete de la aplicación. Para la mayoría de los casos, la variable de Python `__name__` es el valor correcto. Flask usa este argumento para determinar la ruta raíz de la aplicación, para que después pueda encontrar los archivos de los recursos relativos a la ubicación de la aplicación.

Rutas. Los clientes envían peticiones al servidor web, que a su vez las envía a la instancia de la aplicación de Flask. La instancia de la aplicación necesita saber qué código debe regresar para cada URL solicitado, así que mantiene un mapeo de los URL a funciones de Python. La asociación entre el URL y la función que se la maneja se llama *ruta*.

La manera más conveniente de definir una ruta en una aplicación de Flask es a través del decorador `app.route`, que registra la función decorada como una ruta.

El siguiente ejemplo registra la función `index()` como el manejador para el URL raíz de la aplicación. Si la aplicación fue desplegada en un servidor con el nombre del dominio `http://www.lar.com`. Al acceder desde un navegador a la dirección anterior se ejecutará la función `index()` del lado del servidor. El valor retornado por esta función se llama *respuesta*, y es lo que el cliente recibe. Las funciones como `index()` se llaman *funciones de vista*. En este caso el navegador mostrará `Hola mundo`.

```
@app.route('/')
def index():
    return "Hola mundo"
```

Flask soporta los tipos de URL con componentes dinámicos utilizando una sintaxis especial en el decorador `route`. En el siguiente ejemplo se muestra el nombre de usuario como componente dinámico.

```
@app.route('/usuario/<nombre>')
def user(nombre):
    return "Hola, {}".format(nombre)
```

La parte encerrada por los paréntesis angulares es el componente dinámico, así que cualesquiera URL que coincidan con la porción estática será mapeada a esta ruta.

El decorador `app.route` cuenta con un argumento opcional `methods`, el cual recibe una lista de métodos HTTP. `methods` le dice a Flask que registra a las funciones de vista como manejadores para las peticiones de acuerdo al tipo de método enviado.

Cuando `methods` no está en los parámetros de la función, la función de vista es registrada para manejar solamente peticiones GET. El siguiente ejemplo se le pide a Flask que se registre a la función de vista como manejadora de petición de tipo GET y POST.

```
@app.route('/usuario/<nombre>', methods=['GET', 'POST'])
def user(nombre):
    return "Hola, {}".format(nombre)
```

Arranque del servidor. La instancia de la aplicación tiene un método `run` que inicia el servidor web integrado con Flask, el cual sólo está destinado para usarse durante el desarrollo.

```
if __name__ == '__main__':
    app.run(debug=True)
```

Una vez que el servidor es puesto en marcha, entra en un bucle que espera por peticiones para procesarlas. Este bucle continua hasta que la aplicación es detenida.

Existen varios parámetros que `app.run()` puede recibir para configurar el modo de operación del servidor web. Durante el desarrollo, conviene activar el modo de debugging, que entre otras cosas activa el *debugger* y el *reloader*. Esto se hace pasando el argumento `debug` igual a `True`.

Extensiones de Flask

Flask está diseñado para extenderse. Funcionalidades como la autenticación y bases de datos son funciones que el framework deja fuera intencionalmente, dando la libertad de elegir los paquetes que se ajusten mejor a la aplicación.

A continuación se describen las extensiones más importantes usadas en esta aplicación.

Flask-Script. Flask-Script es una extensión que añade un parseador de línea de comandos para tu aplicación de Flask. Un ejemplo de una aplicación en la que se añade el parseo en línea de comandos.

```
from flask import Flask
from flask_script import Manager, Shell

app = Flask(__name__)
```

```
manager = Manager (app)

if __name__ == '__main__':
    manager.run()
```

El método de inicialización de esta extensión es común en muchas otras extensiones: una instancia de la clase principal es inicializada pasando como argumento en el constructor una instancia de la aplicación.

El servidor inicia a través de `manager.run()`, donde la línea de comandos es paraseada.

Si corremos la aplicación anterior en una terminal, esta tiene opciones básicas en la línea de comandos. Correr el programa anterior mostraría un mensaje como el siguiente:

```
$ python app.py
usage: app.py [-h] {shell,runserver} ...

positional arguments:
{shell,runserver}
    shell Runs a Python shell inside Flask application context.
    runserver Runs the Flask development server i.e. app.run()
optional arguments:
-h, --help show this help message and exit
```

El comando `shell` es usado para iniciar un shell de Python en el contexto de la aplicación. Sirve para tareas de mantenimiento, debugging, etc.

El comando `runserver` como su nombre lo dice, inicia el servidor web. Cuenta con varios argumentosopcionales, `--host`, `--port`, `--threaded`, `--no-debug`, entre otros.

Flask-RESTful. Flask-RESTful es una extensión para Flask que añade soporte para la rápida construcción de una API REST. Las características básicas de esta extensión que se necesitan conocer son el *enrutamiento de recursos*, los *endpoints* y el *parseo de argumentos*.

El componente principal que provee Flask-RESTful es la clase `Resource`. Esta brinda un fácil acceso a múltiples métodos HTTP definiendo solamente los métodos en el recurso que se cree.

Se pueden añadir múltiples URL al objeto `Api`, el *punto de entrada principal* de la aplicación, a través de su método `add_resource()`. Cada URL será enrutado al

Recurso que se pase como argumento.

Flask RESTful cuenta con el módulo `reqparse` para la validación de datos en la petición. Sin embargo, a pesar de ayudar con el manejo de errores, no es lo suficientemente robusta para manejar tipos de datos más complejos dentro de la petición.

El siguiente bloque de código muestra una API muy básica, que utiliza todas las características antes mencionadas. Así como la estructura general de un programa usando esta extensión.

```
from flask import Flask
from flask_restful import reqparse, abort, Api, Resource

app = Flask(__name__)
api = Api(app)

TASKS = {
    'tarea1': {'description': 'Busca al chico'},
    'tarea2': {'description': 'Eres el chico'},
    'tarea3': {'description': 'Haz algo por el chico'},
}

def abort_if_task_doesnt_exist(task_id):
    if task_id not in TASKS:
        abort(404, message="Task {} doesn't exist".format(task_id))

parser = reqparse.RequestParser()
parser.add_argument('task')

# muestra, crea, actualiza o elimina una tarea.
class Task(Resource):
    def get(self, task_id):
        abort_if_task_doesnt_exist(task_id)
        return TASKS[task_id]

    def delete(self, task_id):
        abort_if_task_doesnt_exist(task_id)
        del TASKS[task_id]
        return '', 204

    def put(self, task_id):
        args = parser.parse_args()
        task = {'task': args['task']}
```

```

        TASKS[task_id] = task
        return task, 201

# muestra una lista con todas las tareas y permite crear una nueva con POST
class TaskList(Resource):
    def get(self):
        return TASKS

    def post(self):
        args = parser.parse_args()
        task_id = int(max(TASKS.keys()).lstrip('task')) + 1
        task_id = 'tarea{}'.format(task_id)
        TASKS[task_id] = {'description': args['task']}
        return TASKS[task_id], 201

    ## enrutamiento
api.add_resource(TodoList, '/tasks')
api.add_resource(Todo, '/tasks/<todo_id>')

if __name__ == '__main__':
    app.run(debug=True)

```

Flask-SQLAlchemy. Flask-SQLAlchemy simplifica el uso de *SQLAlchemy* dentro de una aplicación de Flask. SQLAlchemy es un poderoso framework para bases de datos relacionales que soporta varios administradores de bases de datos. Ofrece un ORM de alto nivel y acceso de bajo nivel a la funcionalidad SQL nativa de los DBMS.

En Flask-SQLAlchemy, una base de datos se especifica con su URL. Por ejemplo usando *SQLite*, que en realidad es sólo un archivo en el disco, el URL será: **sqlite:///ruta/absoluta/a/la/basededatos**.

El URL de la base de datos añadiendo la llave **SQLALCHEMY_DATABASE_URI** al objeto de configuración de Flask. El ejemplo siguiente muestra como inicializar y configurar una base de datos simple en SQLite.

```

from flask_sqlalchemy import SQLAlchemy
app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] =\
    'sqlite:///data.sqlite'
db = SQLAlchemy(app)

```

El objeto db instanciado de la clase `SQLAlchemy` representa a la base de datos y brinda acceso a toda la funcionalidad de Flask-SQLAlchemy.

El término *modelo* es usado para referirse a entidades persistentes dentro de la aplicación. En el contexto de un ORM, un modelo es una clase de Python con atributos que son iguales a las columnas de la tabla de una base de datos.

La instancia de una base de datos en Flask-SQLAlchemy tiene una clase base para los modelos así como un conjunto de clases y funciones auxiliares que ayudan a definir su estructura. Para el siguiente ejemplo definiremos una base de datos muy simple con una tabla de `Usuarios` y otra de `Recursos`, la relación se define de uno a muchos, cada usuario tiene varios recursos. La Figura 2.3, muestra lo descrito.

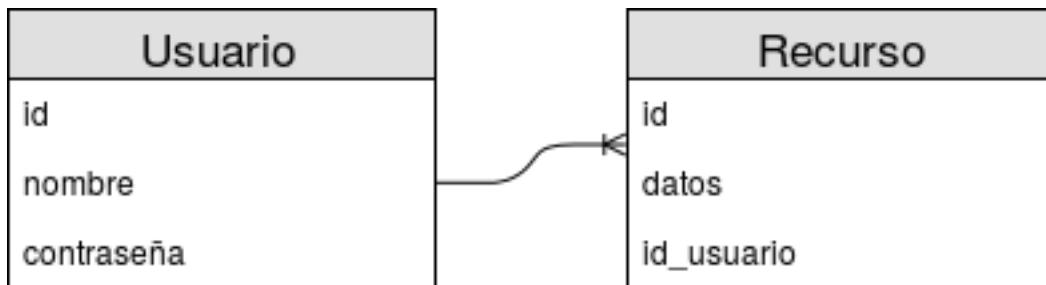


Figura 2.3: Modelo relacional de la tabla usuarios y la de recursos.

El fragmento de código siguiente definen los modelos `Usuario` y `Recurso`.

```
class Usuario(db.Model):
    __tablename__ = 'users'
    id = db.Column(db.Integer, primary_key=True)
    nombre = db.Column(db.String(64), unique=True, index=True)
    password = db.Column(db.String(128))
class Recurso(db.Model):
    __tablename__ = 'resources'
    id = db.Column(db.Integer, primary_key=True)
    data = db.Column(db.Text)
```

El atributo `__tablename__` define el nombre de la tabla en la base de datos. Flask-SQLAlchemy asigna un nombre por defecto si se omite `__tablename__`. Las variables restantes son los atributos del modelo, definidas como instancias de la clase `db.Column`.

El primer argumento en el constructor de `db.Column` es el tipo de dato para la columna de la base de datos. `Integer`, `Float`, `String`, `Text`, son sólo algunos de todas las opciones disponibles. Los argumentos restantes son opciones de configuración para el atributo. Entre las más comunes están `primary_key`, `unique`, `index`, `default` y `nullble`.

Las bases de datos relaciones establecen conexiones entre tuplas de diferentes tablas, con el uso de relaciones. En el diagrama relacional de la Figura 2.3 se establece una relación de *uno a muchos* de usuarios a recursos.

Esta relación uno a muchos se representa en nuestras clases modelo añadiendo un atributo más, en la clase `Recurso` agregamos como llave foránea el id del usuario al que le pertenece `id_usuario = db.Column(db.Integer, db.ForeignKey('usuarios.id'))`. Para el modelo de `Usuario` agregamos el atributo `recursos = db.relationship('Recurso', backref='usuario')`. El primer argumento indica que modelo está del otro lado de la relación, `backref` establece la relación en la dirección opuesta.

Hay otros tipos de relaciones. La relación *uno a uno* puede ser expresada como la relación anterior pero agregando la bandera `uselist` a `db.relationship()`, que indica un atributo escalar en vez de una colección del lado de los muchos en la relación.

Operaciones sobre la base de datos

- Crear tablas: `db.create_all()`. Crea la base de datos basándose en las clases modelos.
- Borrar tablas: `db.drop_all()`. Destruye los datos en la base de datos.
- Añadir tuplas: `db.session.add(Objeto)`. Los cambios se manejan a través de una `session` en la base de datos. Se pasa como parámetro un objeto del modelo.
- Escribir cambios: `db.session.commit()`. Para guardar los objetos en la base de datos, las sesiones necesitan ser *confirmadas*.
- Modificar tupas: `Objeto.atributo_a_actualizar = nuevo_valor db.session.add(Objeto)`. El método `add` también se puede usar para actualizar los modelos.
- Eliminar tuplas: `db.session.delete(Objeto)`. El método `delete` recibe como parámetro al objeto que desea eliminarse.
- Consultas: `Modelo.query.all()` `Modelo.query.filter_by(Criterio)`. La consulta más básica retorna todo el contenido de la tabla correspondiente `Modelo.query.all()`. Se pueden usar *filtros* para buscar elementos más específicos en la base de datos.

Requests. Requests permite hacer peticiones HTTP de una manera muy sencilla. Un petición de tipo GET se convierte en algo simple como lo siguiente:

```

>>> import requests
>>> r = requests.get('https://swapi.co/api/planets/1/')
>>> r.status_code
200
>>> r.headers['content-type']
'application/json'
>>> r.text
u'{"name": "Tatooine", "rotation_period": 23, "orbital_period": 304, ^
  "diameter": 10465, "climate": "arid", ...}'
>>> r.json()
{u'diameter': u'10465', u'climate': u'arid', u'surface_water': u'1', u
  'name': u'Tatooine', ...}

```

Hay varios métodos para hacer una petición, sin embargo, nos enfocaremos en uno. El método `request` de la clase `requests`, construye, envía una petición y tiene como valor de retorno un objeto `Response`. A continuación de los parámetros más comunes del método `requests.request(method, url, **kwargs)`

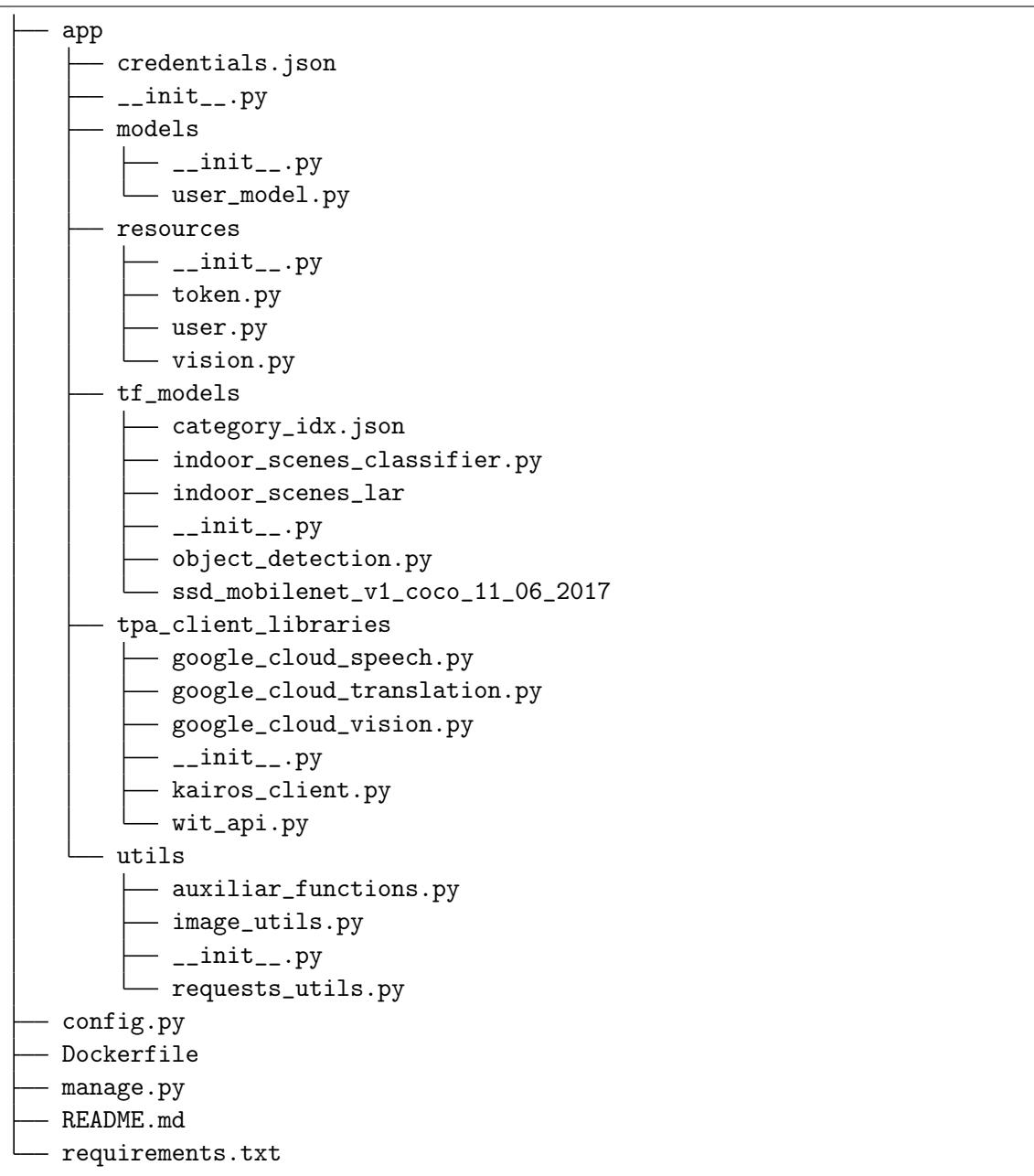
- `method`, (obligatorio) el método de la petición, una cadena con valor `post`, `get`, etc.
- `url`, (obligatorio) la URL a la que se hace la petición.
- `params`, (opcional) diccionario o bytes enviados en la URL.
- `data`, (opcional) diccionario o lista de tuplas `[(llave, valor)]`, bytes u objeto similar a un archivo.
- `json`, (opcional) json a enviar en el cuerpo de la petición.
- `headers`, (opcional) un diccionario con los headers HTTP que se enviarán.

Dependiendo del contenido de la respuesta por parte del servidor, se elige cómo procesar la respuesta. Si es solo texto, se puede usar `Response.text`, que automáticamente decodificará el contenido enviado por el servidor, `Response.content`, si queremos acceder al cuerpo del mensaje como bytes. También cuenta con un decodificador de JSON, `Response.json()`.

Podemos checar el código de estado de la respuesta con `Response.status_code`. Y también los headers, utilizando `Response.headers`.

Descripción de los módulos principales

La aplicación está compuesta por diferentes por módulos divididos de la siguiente forma:



En el nivel más alto están:

- La aplicación de Flask en un paquete llamado *app*. Este es el núcleo del proyecto, contiene los modelos para la base de datos, los recursos, clientes de servicios web de terceros, etc.
- El archivo que almacena las opciones de configuración *config.py*.
- El script que ejecuta al aplicación *manage.py*.

- Los archivos *README.md* y *requirements.txt*, y la carpeta *venv*, contienen una breve descripción del proyecto, la lista de dependencias y el ambiente virtual de Python, respectivamente.

En las siguientes secciones se documentan las partes más importantes del proyecto, el paquete *app*, *config.py* y *manage.py*.

Aplicación (app). La aplicación en Flask de la API de CloudNAO, está compuesta por varios paquetes, los recursos, los modelos para la base datos, los clientes de los servicios web de terceros, modelos de Tensorflow y módulos auxiliares.

Modelos (models). Este paquete contiene los modelos de la base de datos, usando SQLAlchemy.

```
class app.models.user_model.UserModel(username, password)
```

Una clase pública para el modelo del usuario que se almacena en la base de datos. El nombre de la tabla en la base de datos es **users**. Los atributos de esta última se describen en la tabla 2.3.

Tabla 2.3: Descripción de los atributos del modelo User

Atributo	Descripción
id	El id del usuario, es una llave primaria.
username	Una cadena con el nombre del usuario, es única.
pass-word_hash	Una cadena que obtenida de una función hash aplicada a la contraseña original
ac-cess_token	Una cadena única que sirve como llave de autorización al hacer peticiones a los recursos de la API.

El siguiente ejemplo muestra la creación e inserción de un usuario en la base de datos.

```
>>> from app.models.user_model import UserModel
>>> rick_deckard = UserModel('rick.deckard@sfpd.gov', '4m1r34l2019')
>>> rick_deckard.username
'rick.deckard@sfpd.gov'
>>> rick_deckard.password_hash
'pbkdf2:sha256:50000$oS...NfC6Y1$e54e06fcfc0a608795fe32147e4a... '
>>> rick_deckard.access_token
'Z9WxxgXeYxI0g8N6hD6Jj4UM79fJTf4uAR3VAUyFUbXWK8k14D'
>>> UserModel.query.all()
```

```
[]  
>>> rick_deckard.save_to_db()  
>>> UserModel.query.all()  
[<UserModel 1>]
```

Recursos (*resources*). En este paquete se definen los recursos que representan a los servicios que la API estará brindando. Desde el registro de un usuario, obtención de un nuevo token de acceso, hasta el procesamiento de imágenes.

```
class app.resources.user.UserRegister
```

Una clase pública que representa un recurso de la API REST, para el almacenamiento de nuevos usuarios. Hereda de la clase `Resource` de `flask_restful` y usa un objeto de la clase `RequestParser` para definir y parsear los argumentos de cada petición HTTP. Sólo se define el método POST, para la creación de usuarios.

Son dos los argumentos requeridos para poder solicitar este recurso, `username`, el nombre del usuario, y `password`, la contraseña. Si alguno de estos dos no es enviado, la instancia de `RequestParser` envía un mensaje de error.

`post()`

Método para añadir un nuevos usuario a la base de datos, a través de un POST. Se verifica si no hay un usuario con el mismo nombre. Si existe, se envía un mensaje de error y el código de estado *400 (Bad Request)*. Si no, se envía un mensaje de éxito y código de estado *201 (Created)*.

Un ejemplo del cuerpo de la petición y la respuesta se muestran a continuación:

Petición

```
{  
    "username" : "rick.deckard@sfpd.gov"  
    "password" : "4m1r34l2019"  
}
```

Respuesta

```
{  
    "message" : "User created successfully."  
    "token" : "Z9WxxgXeYxI0g8N6hD6Jj4UM79fJTf4uAR3VAUyFUbXWK8kl4D"  
}
```

```
class app.resources.token.Token
```

Clase pública que representa un token en la API REST. Permite la obtención de un nuevo token de acceso a la API, usando un método POST y enviado en el mensaje el nombre de usuario y la contraseña. Todo esto usando Resource y RequestParser de flask_restful.

Solo se define el método POST, para generar un nuevo token de acceso.

```
post()
```

Genera un nuevo token de acceso para el usuario que envía un POST a este recurso. Se requiere enviar el nombre del usuario y la contraseña. Si estos son válidos se genera un nuevo token y se guarda en la base de datos. Se envía como respuesta el nuevo token, con un código de estado *201 (Created)*.

El cuerpo de la petición y la respuesta, si todo es exitoso, son los siguientes:

Petición

```
{  
    "username" : "rick.deckard@sfpd.gov"  
    "password" : "4m1r34l2019"  
}
```

Respuesta

```
{  
    "token" : "ue6yCsfMBX3D0oMhieBgbY6eA12tDWsZkOnX000qP0qglWJ6Kq"  
}
```

```
class app.resources.vision.Vision
```

Clase pública del recurso de la API REST para solicitar el procesamiento de imágenes. Acepta peticiones de tipo POST en cuyo cuerpo van las características que se desean computar. Las aceptadas son las siguientes:

- FACE_RECOGNITION
- OBJECT_DETECTION
- LABELS_DETECTION
- OCR_TRANSLATION
- FACE_ENROLL

- CLASSIFY_INDOOR_SCENES

post()

Define el método POST a este recurso. Requiere de enviar un token de acceso en los encabezados para poder ejecutarse. Solamente maneja la petición y corre la funciones que se piden de acuerdo a lo que se solicita en el cuerpo del mensaje.

Aquí se unen en una sola las respuestas de los módulos dentro de app.tf_models y app.tpa_client_libraries.

Modelos de Tensorflow (*tf_models*). En este paquete se encuentran los módulos encargados de la detección de objetos y la clasificación de escenarios usando TensorFlow.

```
class app.tf_models.object_detection.ObjectDetectionTensorflow
```

Una clase pública que utiliza un modelo pre-entrenado para la detección de múltiples objetos en una imagen. El modelo utilizado es ssd_mobilenet_v1_coco provisto por la API de TensorFlow y entrenado con el conjunto de imágenes de COCO.

El siguiente ejemplo muestra un simple uso de la clase.

```
>>> from app.tf_models.object_detection import ObjectDetectionTensorflow
>>> object_detector = ObjectDetectionTensorflow()
>>> image_base64 = "/9j/4AAQSkZJRgABAQEAYABgAAD/2... "
>>> object_detector.object_detection(image_base64)
[{'confidence': 0.8984639048576355, 'category': 'car', 'topLeftY': 175.0,
 ↳'height': 129.0, 'width': 154.0, 'topLeftX': 0.0}, ..., {'confidence': 0.
 ↳6006952524185181, 'category': 'person', 'topLeftY': 189.0, 'height': 109.0,
 ↳'width': 45.0, 'topLeftX': 164.0}]
>>>
```

```
class app.tf_models.indoor_scenes_classifier.ImageClassifier
```

Una clase pública que utiliza que utiliza una red neuronal convolucional implementada en Tensorflow para clasificar en cuatro clases una imagen. Las clases son: desks, exit, office, y soccer_court.

Es la clase que carga el modelo ya entrenado, simplemente carga la gráfica de cómputo que diseñamos con los parámetros aprendidos. El clasificador puede recibir una imagen codificada en base64 o su URL. El resultado es un diccionario que pueda ser enviado como respuesta en la API REST.

El siguiente ejemplo muestra un simple uso de la clase. La imagen utilizada es la de la figura 2.4 codificada en base 64.

Figura 2.4: Cancha de entrenamiento del robot NAO.



```
>>> from app.tf_models.indoor_scenes_classifier import ImageClassifier  
>>> test_image = "/9j/4AAQSkZJRgABAQAAAQABAAAD/2..."  
>>> clasifier = ImageClassifier()  
>>> classifier.classify_image(test_image)  
{'indoor_scene': 'soccer_court'}  
>>>
```

Cliente de servicios web de terceros (*tpa_client_libraries*) En este paquete se encuentran los módulos para hacer las peticiones a las API REST de los servicios en la nube de Google Cloud, Kairos y Wit.ai.

```
class app(tpa_client_libraries.google_cloud_translation.  
GoogleCloudTranslation
```

Una clase pública que se conecta a la API de traducción de Google Cloud, la cual permite traducir una cadena dada en un cualquier idioma admitido. Hay tres métodos que provee la API; `translate`, que recibe una cadena y retorna su traducción, `detect`, para detectar el idioma de un texto y `languages`, que retorna la lista de todos los idiomas disponibles para la traducción.

El usado es `translate`, a través de un POST a <https://translation.googleapis.com/language/translate/v2>. El cuerpo del mensaje HTTP debe tener al menos dos parámetros `q` y `target`, el texto de entrada y el código del lenguaje al que se traducirá el texto de entrada, respectivamente.

En el siguiente fragmento de código en una terminal de Python interactiva se muestra cómo utilizar este módulo.

```

>>> from app.tpa_client_libraries.google_cloud_translation import GoogleCloudTranslation
>>> translate = GoogleCloudTranslation()
>>> translate.translate("Cuervo, cuervo, no soporto más tu mirada", "en")
{'sourceLanguage': 'es', 'targetText': 'Crow, crow, I can not stand your gaze anymore', 'sourceText': 'Cuervo, cuervo, no soporto más tu mirada'}
>>>

```

```

class app.tpa_client_libraries.google_cloud_vision.
GoogleCloudVision

```

Una clase pública para integrar en la aplicación dos de las características ofrecidas por Google Cloud Vision, el reconocimiento óptico de caracteres y el etiquetado de imágenes, usando su API REST.

El recurso de la API de Google Cloud Vision utilizado es `v1.images`, y la URL es <https://vision.googleapis.com/v1/images:annotate>. Por lo que hacer un POST al URL anterior ejecuta el procesamiento de un lote de imágenes.

El cuerpo del mensaje HTTP en la petición a la API de Google Cloud Vision, debe tener la siguiente estructura.

```

{
  "requests": [
    {
      "features": [
        {
          "type": "TEXT_DETECTION"
        },
        {
          "type": "LABEL_DETECTION"
        }
      ],
      "image": {
        "content": "",
        "source": {
          "imageUri": ""
        }
      }
    }
  ]
}

```

En el parámetro `features` solo añadimos las dos que utilizamos. El objeto `image`

contiene la imagen que se desea analizar, y acepta la imagen codificada en base64 (`content`) o un URL (`imageUri`).

Además de el cuerpo con la imagen es necesaria una API Key en los parámetros del URL.

El siguiente ejemplo muestra un uso simple de esta clase cliente de la API de Google Cloud.

```
>>> from app.tpa_client_libraries.google_cloud_vision import
    ↵GoogleCloudVision
>>> vision = GoogleCloudVision()
>>> vision.text_annotations_description('https://new2.fjcdn.com/comments/
    ↵Quote+from+this+scene+in+blade+runner+1982+dayum+son+
    ↵fbf2a406e9c802ff492e414c34fff791.jpeg', True)
{'text': 'You reach down and you flip the\\ntortoise over on its back,
    ↵Leon.\n'}
>>> vision.label_annotations_description('https://new2.fjcdn.com/comments/
    ↵Quote+from+this+scene+in+blade+runner+1982+dayum+son+
    ↵fbf2a406e9c802ff492e414c34fff791.jpeg', True)
[{'confidence': 0.76214135, 'name': 'photo caption'}, {'confidence': 0.
    ↵569268, 'name': 'film'}, {'confidence': 0.55112684, 'name': 'screenshot
    ↵'}]
>>>
```

```
class app.tpa_client_libraries.kairos_client.Kairos
```

Una clase pública que sirve como cliente para conectarse a dos servicios que brinda la API de Kairos, el almacenamiento de rostros en una galería en la nube, y la identificación de caras previamente guardadas.

Para ocupar estos servicios la API de Kairos provee los recursos `enroll` y `recognize`, los cuales soportan el método `POST` y tienen como URL base `http://api.kairos.com/`. Para hacer estas peticiones se necesita una forma de autenticación, y esta es a través de los encabezados del mensaje HTTP, enviando en estos un id y una llave de la aplicación.

El cuerpo del `POST` a `recognize` tiene necesita al dos parámetros; la imagen, que puede ser su URL o la codificación en base 64 y el nombre de la galería de donde de donde se desea identificar el rostro.

```
{
    "image" : "",
    "gallery_name" : ""
}
```

El cuerpo del POST a enroll requiere de al menos tres argumentos; la imagen, el nombre de la galería y el identificador del rostro, que comúnmente es el nombre de la persona con esa cara.

```
{  
    "image" : "",  
    "subject_id" : "",  
    "gallery_name" : ""  
}
```

Un ejemplo del uso del módulo.

```
>>> from app.tpa_client_libraries.kairos_client import Kairos  
>>> face = Kairos()  
>>> face.enroll("http://www.indiewire.com/wp-content/uploads/2017/10/  
→triboro_build03.jpeg?w=780", "Rachael")  
{'faceEnroll': {'gender': 'M', 'width': 235, 'topLeftY': 232, 'confidence  
↳': 0.99931, 'height': 235, 'topLeftX': 232}}  
>>> face.recognize("https://i.ytimg.com/vi/j4jIJB8c1I8/maxresdefault.jpg")  
{'faceRecognize': {'subject_id': 'Rachael', 'width': 342, 'topLeftY': 165,  
↳ 'confidence': 0.63092, 'height': 342, 'topLeftX': 366}}  
>>>
```

```
class app.tpa_client_libraries.wit_api.WitAPI
```

Una clase pública donde se hacen las peticiones HTTP a Wit.ai y para extraer el significado del audio o texto. Funciona como un módulo cliente para solicitar los recursos de /message y /speech. Estos recursos tienen la función de hallar información estructurada dentro de una oración.

En seguida se muestra cómo utilizar el método message() para enlistar las entidades encontradas en una cadena de texto.

```
>>> from app.tpa_client_libraries.wit_api import WitAPI  
>>> wit_example = WitAPI()  
>>> wit_example.message("Lee el texto de esta fotografía")  
{'textDetect': [{}{'confidence': 0.76588128565519, 'type': 'value', 'value  
↳': 'lee'}, {}{'confidence': 0.95853105168152, 'type': 'value', 'value':  
↳ 'texto'}], 'photography': [{}{'confidence': 0.65220641878769, 'type':  
↳ 'value', 'value': 'fotografía'}]}  
>>>
```

Utilidades (*utils*) Este paquete incluye funciones auxiliares para hacer peticiones HTTP utilizando la biblioteca `requests` de Python, funciones para validar la autenticación y cargar una imagen en un arreglo de `numpy`.

Configuración y ejecución

La aplicación cuenta con varias opciones de configuración, principalmente para `SQLAlchemy` y para activar el debugging durante la ejecución.

Para la ejecución de la aplicación simplemente se corre el script `manage.py`. Eligiendo ya sea la interfaz de línea de comandos:

```
$ python manage.py shell
```

O lanzar el servidor:

```
$ python manage.py runserver
```

config.py Programa que establece las opciones de configuración para la aplicación.

manage.py El programa que inicia la aplicación.

2.1.3. Firebase Realtime Database y CloudNAO

Como componente dentro de la arquitectura CloudNAO, la Realtime Database tiene dos funciones principales; almacenar un registro de datos generados por el robot, y servir como interfaz para enviar desde un navegador web comandos que puedan ser ejecutados por el robot de manera remota, sin la necesidad de estar conectados a la misma red. A la primera funcionalidad la llamaremos *Robot Log* y a la segunda *Robot Remote Control*.

Firebase Realtime Database es el elemento que relaciona al robot, la aplicación web y la móvil. En la Figura 2.5 se muestran las relaciones. La unión del robot, la aplicación móvil y Firebase Realtime Database forma parte de la función de *Robot Log*. La conexión entre el robot, la aplicación web y Firebase Realtime Database son la otra parte del *Robot Log*, y toda la estructura del *Robot Remote Control*.

De la Figura 2.5 se muestran identificadas por un número las relaciones entre cada elemento. A continuación se describe cada una de estas:

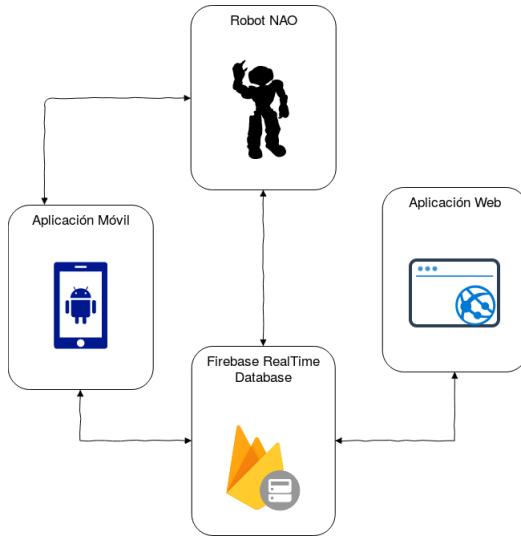


Figura 2.5: Diagrama de la relación entre los componentes de la arquitectura y Firebase Realtime Database

1. **Robot - Firebase Realtime Database**, el robot envía información a la base de datos mediante su API REST. Para la parte de *Robot Log*, envía datos de **ALMemory** o de **ALVideoDevice** y para *Robot Remote Control* establece una conexión constante para recibir actualizaciones que son interpretadas como comandos. La API REST soporta el protocolo Server-sent events, la cual es una tecnología donde un cliente recibe actualizaciones automáticamente desde el servidor mediante una conexión HTTP. Con esta herramienta el robot se mantiene «escuchando» los cambios realizados a una ubicación en la base de datos. Al recibir un mensaje el robot lo maneja como un comando para ejecutar cierta acción, como puede ser un movimiento o enviar una imagen en vivo. Esta es la parte principal del *Robot Remote Control*.
2. **Aplicación móvil - Robot**, utilizando el SDK de NAOqi para Java, específicamente para Android, se ejecutan remotamente módulos de la API de NAOqi. Para la parte de Realtime Database se solicitan datos de **ALMemory**. El robot envía la información a la aplicación y ésta escribe los datos en una ubicación única para cada robot. Esta relación es una de las partes que generan los datos de *Robot Log*.
3. **Firebase Realtime Database - Aplicación móvil**, el primer elemento envía los últimos valores guardados en la base de datos a una lista que se muestra dentro de una actividad en la aplicación cuya función es mostrar valores obtenidos de **ALMemory**, esta parte corresponde a *Robot Log*. Usando el SDK para Android de Firebase Realtime Database se hacen realizar las operaciones de escritura y lectura de los datos de **ALMemory** obtenidos del robot.
4. **Firebase Realtime Database - Aplicación web**, el navegador web escribe

y lee los datos con el SDK web de Firebase Realtime Database. Se vale de la funcionalidad de *Robot Log* para mostrar los últimos valores de ALMemory enviados por el robot y escribe información a una ubicación de la base de datos a la que el robot se habrá suscrito para detectar cambios. Dichos cambios son comandos que el robot ejecutará. Esto último es una parte de *Robot Remote Control*.

En las secciones subsecuentes se explica cómo implementar el servicio de Firebase Realtime Database en cada una de las plataformas utilizadas, el robot NAO, la aplicación móvil y la web.

Diseño de la base de datos

Es muy importante estructurar de manera correcta la información en una base de datos de Firebase.

Los datos dentro de la Firebase Realtime Database se almacenan como objetos JSON. La base de datos puede conceptualizarse como un árbol JSON alojado en la nube. A diferencia de una base de datos de SQL, no hay tablas ni registros. Cuando le agregas datos al árbol JSON, estos se convierten en un nodo de la estructura JSON existente con una clave asociada.

La documentación de Firebase recomienda algunos puntos que se deben tomar en cuenta para estructurar los datos dentro del JSON.

- **Evitar anidación de datos.** Firebase Realtime Database permite anidar datos en un máximo de 32 niveles, sin embargo, cuando obtienes datos de una ubicación de la base de datos, también se recuperan todos los nodos secundarios.
- **Compactar las estructuras de datos.** Si los datos se dividen en rutas de acceso independientes, que también se conocen como no normalizadas, se pueden descargar de manera eficaz en llamadas separadas, según sea necesario.
- **Crear datos escalables.** Siempre es mejor desgar un subconjunto de una lista de registros.

Siguiendo estas recomendaciones, se diseñó una base de datos, evitando anidar elementos separándolos como fuera conveniente y manteniendo relaciones que eran necesarias para realizar consultas de manera simple.

En las siguientes secciones primero se describe la base de datos con un enfoque clásico con el modelo entidad relación y después como se puede transformar ese modelo base en un JSON que se pueda almacenar en la nube.

El diagrama de la Figura 2.6, muestra el modelo sobre el que se basa el JSON que se almacena en Firebase.

Un usuario puede tener varios robots, cada robot cuenta está relacionado con otras entidades como: un texto que representa el discurso que tiene que decir, la última imagen enviada por el robot, un largo historial de logs con valores de ALMemory, el estado del robot (el nivel de batería y su conexión a la red), y comandos para ejecutar ciertas tareas.

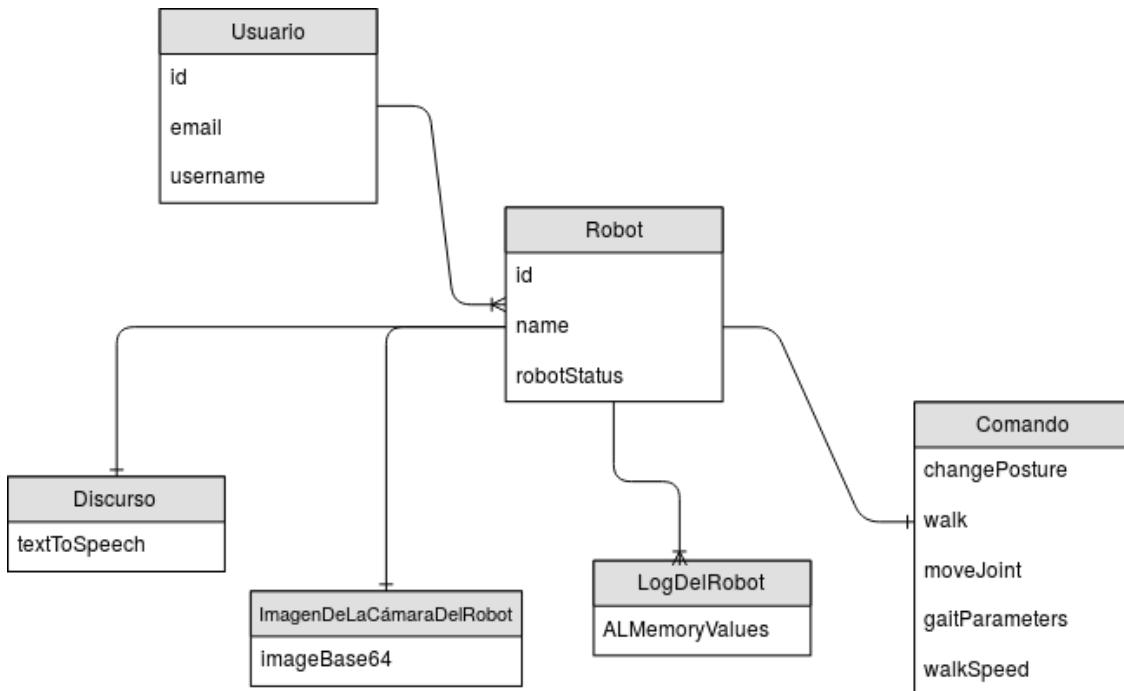


Figura 2.6: Diagrama entidad relacin de la base de datos para Firebase

Estructura del JSON para Firebase

La estructura de la base de datos descrita en el modelo de la Figura 2.6 es la base del JSON y es por eso que puede no cumplir los requisitos de normalización, simplemente ilustra el diseño de la base de datos para que sea más simple su visualización.

Con una idea de cómo está diseñada la base de datos la estructura del JSON queda como en la Figura 2.7. Es importante señalar que esta base de datos es muy flexible, los campos en las «entidades» que aquí son los atributos del JSON, pueden cambiar: eliminándolos, agregando nuevos, etcétera.

Para entender mejor el JSON de la Figura 2.7, se describen algunos de sus principales atributos:

- **nao-firebase**, es la raíz del objeto.

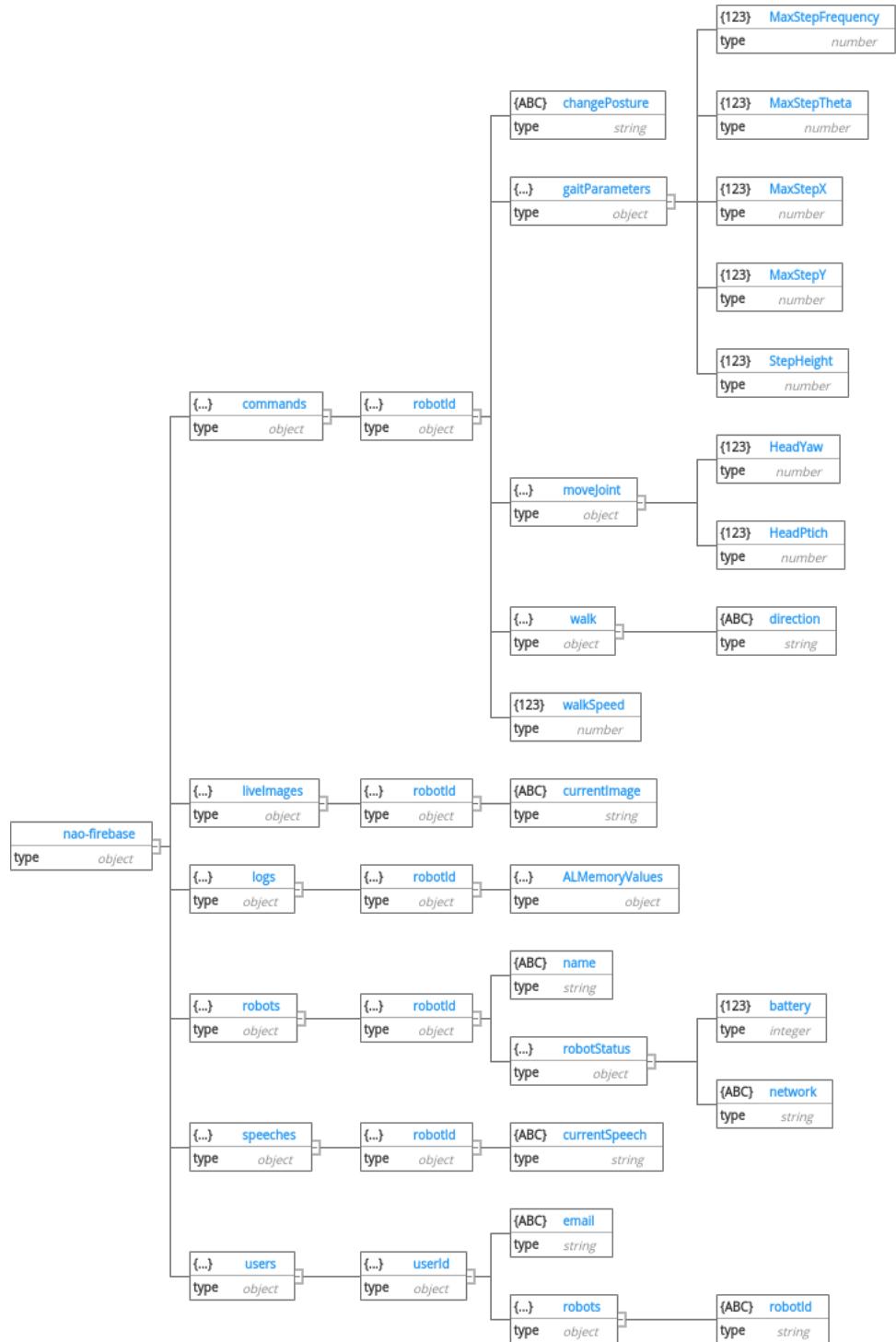


Figura 2.7: La estructura del JSON donde se almacenan los datos

- **commands**, es parte del segundo nivel, los robots se suscriben esta ubicación para manejar actualizaciones. Está compuesto por objetos que corresponde a cada robot.
- **liveImages**, es parte del segundo nivel, y almacena imágenes enviadas por cada robot. Está compuesto por objetos que corresponde a cada robot.
- **logs**, está en el segundo nivel, y aquí se encuentra el historial de logs de **ALMemory** de cada robot. Está compuesto por objetos que corresponde a cada robot.
- **robots**, definido en el segundo nivel, conjunto de todos los robots que se añaden a la aplicación.
- **speeches**, está en el segundo nivel y contiene el texto que debe ser enviado al robot y éste repetir oralmente. Está compuesto por objetos que corresponde a cada robot.
- **users**, en el segundo nivel, el conjunto de todos los usuarios de la aplicación, ordenados por su identificador generado por Firebase Auth. Sólo se almacena su dirección de correo electrónico y un objeto con los robots que le pertenecen.
- **robotId**, es un identificador único generado por Firebase, cada robot que agrega un usuario tiene un id único. Es una cadena que sirve para que cada robot tenga su propia ubicación de comandos, imágenes, etc.
- **userId**, un id obtenido de Firebase Authentication.

Los demás atributos son autodescriptivos.

2.1.4. Aplicación para Android CloudNAO

Esta aplicación móvil es el componente dentro de la arquitectura CloudNAO que permite interactuar con el robot sin necesidad de instalar software en el robot. Se conecta con el robot usando el SDK para Android de NAOqi, hace peticiones a la API RESTful de CloudNAO, y además utiliza la autenticación y la base de datos en tiempo real de Firebase utilizando su SDK para Android.

Es una aplicación moderna que no se limita a ser un simple control remoto, y añade funcionalidades como la escritura de logs generados por el robot en la nube, y el potente análisis de imágenes simplemente haciendo peticiones a la API RESTful de CloudNAO.

En las siguientes secciones se describe con mayor detalle cada elemento de la aplicación, ya sea para la parte que ve el usuario final como los componentes internos para que desarrolladores puedan mantenerla o añadir nuevas funcionalidades.

Descripción de la aplicación

Requisitos

La aplicación fue probada con éxito sobre dispositivos con arquitecturas ARM, y con una versión menor o igual a Android 5.1.1 (API 22). Los problemas con nuevas plataformas son que el SDK provisto por Aldebaran, no es compatible con arquitecturas x86, ni con nuevas versiones de Android. Esta incompatibilidad se debe a la forma en que se compiló el archivo *java-naoqi-sdk-2.1.4-android.jar*.

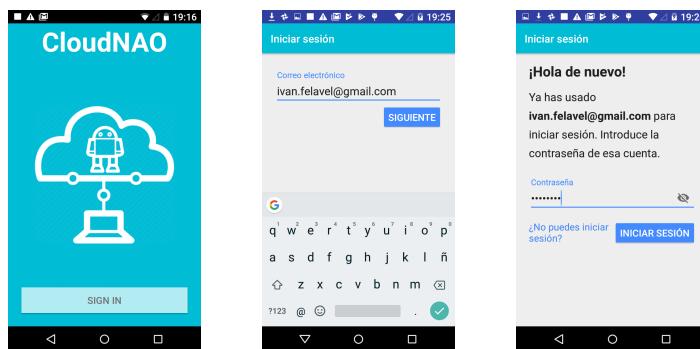
Funcionalidades

La aplicación ofrece algunas características que más que nada sirven para mostrar casos de uso para algunos recursos dentro de la API RESTful de CloudNAO, y para utilizar productos de Firebase como la autenticación o la base de datos en tiempo real.

Cabe señalar que al usar a Firebase como BaaS, la aplicación móvil comparte información con la aplicación web de CloudNAO. Por ejemplo, los usuarios que se registren en una u otra, pueden iniciar sesión en cualquiera sin problemas; similar es el caso de los robots que registren y la información de cada uno.

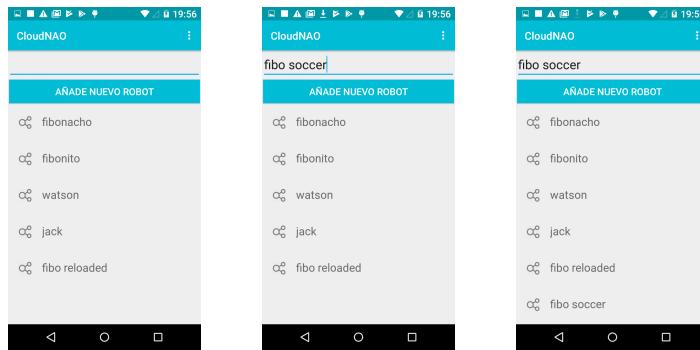
En general las siguientes son las funcionalidades ofrecidas por las aplicación.

Autenticación de usuarios. Para poder ocupar la aplicación los usuarios deben iniciar sesión con un correo y contraseña. La aplicación cuenta con la opción de registro, si el usuario es nuevo.

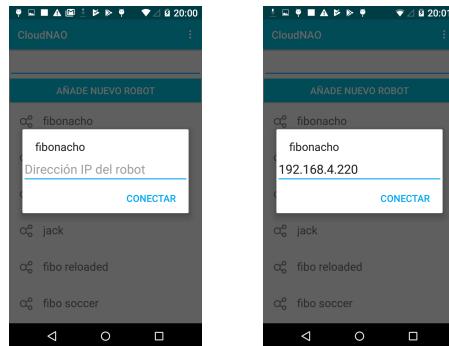


Selección y adición de robots a la aplicación. Para llevar un control de los robots que tiene un usuario se le permite añadir los robots que sean necesarios.

Además el usuario cuenta con una lista que le muestra los robots disponibles que previamente creó.



Conexión con un robot seleccionado. Al elegir un robot entre los que el usuario posee, simplemente escribe la dirección IP del robot, y se crea una conexión con éste dentro de la misma red. Con esto, es posible acceder a todas las características disponibles en la aplicación.



Menú intuitivo. Las actividades que se pueden realizar después de conectar la aplicación con el robot se muestran en un menú intuitivo.

Control remoto. Una primera necesidad que se encontró y por la que surgió todo el proyecto fue crear un control remoto para el robot NAO. Es muy básico pero admite comandos para hacer al robot caminar sobre sus tres ejes y cambiar entre dos posturas. El control remoto incluye una imagen en vivo de la cámara del robot.

Figura 2.8: La pantallas del menú de la aplicación.

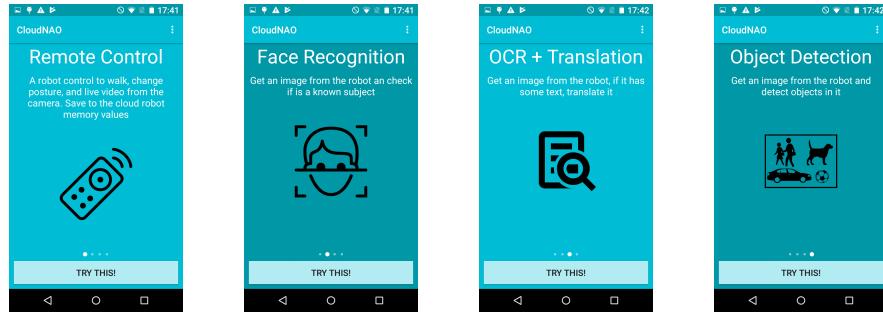
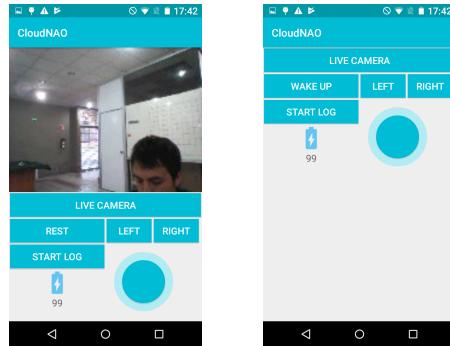


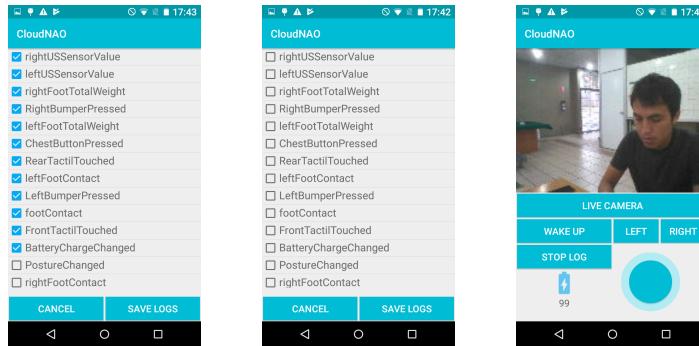
Figura 2.9: El control remoto muestra una imagen en vivo si se desea, el estado del batería y un joystick para mover al robot.



Guarda valores de ALMemory en la nube. Otra función dentro del control remoto es la de la opción de guardar ciertos valores de la memoria del robot en la nube, para consultarlos o descargarlos después en la aplicación web. El conjunto de valores que por ahora están disponibles para su almacenamiento son los siguientes: rightUSSensorValue, leftUSSensorValue, rightFootTotalWeight, RightBumperPressed, leftFootTotalWeight, ChestButtonPressed, RearTactilTouched, leftFootContact, LeftBumperPressed, footContact, FrontTactilTouched, BatteryChargeChanged, PostureChanged, rightFootContact, MiddleTactilTouched, GyrometerX, GyrometerY, AccelerometerX, AccelerometerY, AccelerometerZ, TorsoAngleX, TorsoAngleY.

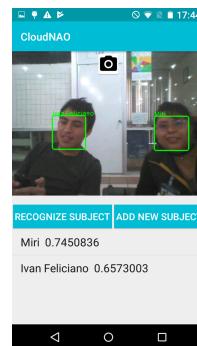
Detección de rostros en una fotografía enviada por el robot. La detección de rostros es un problema levemente resuelto por el equipo de Aldebaran; existe un módulo dentro de NAOqi para seguir rostros o detectarlos, así como reconocer algunos antes guardados. Sin embargo, los modelos de visión que utilizan no son lo suficientemente potentes, ya que deben ejecutarse dentro del robot. Esta fun-

Figura 2.10: Después de dar clic en el botón START LOG se abre una pantalla para elegir los valores de la memoria que se desean guardar en Firebase.



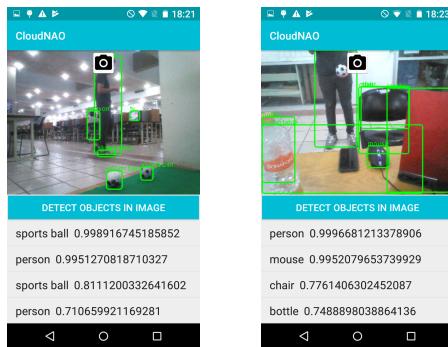
cionalidad dentro de la aplicación, muestra la facilidad con la que se puede hacer procesamiento de imágenes con solo una petición HTTP. El usuario simplemente toma una fotografía con la cámara del robot, añade la etiqueta de la persona que se encuentra en esa imagen y esto se envía a la API de CloudNAO que a la vez se conecta con la de Kairos. Si todo fue exitoso un nuevo rostro se conserva para su futuro reconocimiento.

Reconocimiento de rostros previamente almacenados. Es la continuación de la característica mencionada anteriormente, reconoce a sujetos antes guardados a través de una foto de sus rostros. Cuando la aplicación ha detectado los rostros y reconocido, brinda la posibilidad de que el robot ejecute una animación y diga un breve saludo a cada persona encontrada.

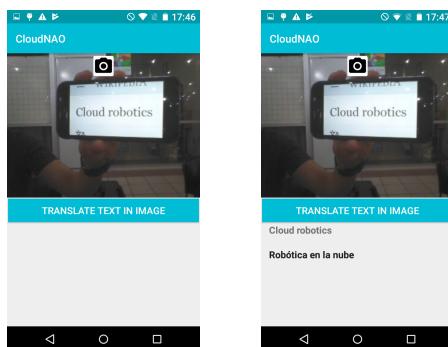


Detección de objetos de entre 80 categorías en una imagen de la cámara del robot. Funciona de manera similar a la detección de rostros. Envía una imagen capturada con la cámara del robot a la API RESTful de CloudNAO, que ejecuta un módulo con la API de detección de objetos de Tensorflow y se procesa

la respuesta para que en la pantalla se dibujen unos recuadros delimitadores sobre los objetos detectados así como una etiqueta con el nombre que le corresponde. El robot simplemente dice la cantidad de objetos que ve de acuerdo a las categorías válidas.



Reconocimiento óptico de caracteres y traducción de texto encontrados en una imagen enviada por el robot. La aplicación solicita una imagen del robot, esa imagen es parte de la petición a la API RESTful de CloudNAO, igual que en las dos funcionalidades anteriores. Se buscan caracteres dentro de la imagen, para su posterior traducción. El resultado se muestra en dos partes, una es el texto original y el otro es el texto traducido al español. El robot tiene la opción de convertir la traducción en un discurso oral.



Guía para desarrolladores

Para desarrollar la aplicación fueron tres los elementos importantes:

- Un entorno de desarrollo integrado (**Android Studio**).
- El robot humanoide NAO (no simulado).

- El SDK para Android de NAOqi.

Además de los elementos anteriores se necesitan una computadora y acceso a una red inalámbrica en cada uno de los dispositivos.

En las siguientes subsecciones se describen las herramientas ocupadas durante el desarrollo de la aplicación: Android Studio, Firebase Realtime Database, Firebase UI para la autenticación, y las bibliotecas Butterknife y Volley para el manejo de la IU y para las peticiones a la API REST, respectivamente. Después de la breve descripción de los elementos mencionados se listan las actividades principales (subclases de `Activity`) dentro de la aplicación móvil.

Android Studio

Android Studio es el entorno de desarrollo integrado (IDE por sus siglas en inglés) oficial para desarrollar sobre la plataforma Android, está basado en el popular IntelliJ IDEA.

La instalación de Android Studio incluye:

- El Android SDK
- Herramientas del Android SDK y herramientas de plataforma. Instrumentos para el debugging y el testing de tus aplicaciones.
- Una imagen del sistema para el emulador de Android. Permite crear y probar aplicaciones sobre dispositivos virtuales diferentes.

La versión de Android Studio utilizada en este proyecto fue la 2.3.3. Probablemente existan problemas de compatibilidad con nuevas versiones.

Estructura de un proyecto

Cada proyecto en Android Studio contiene uno o más módulos con archivos de código fuente y archivos de recursos. Entre los tipos de módulos se incluyen los siguientes:

- módulos de apps para Android
- módulos de bibliotecas

De manera predeterminada, Android Studio muestra los archivos de tu proyecto en la vista de proyectos de Android, como se muestra en la figura 1. Esta vista se organiza en módulos para proporcionar un rápido acceso a los archivos de origen clave de tu proyecto.

Cada módulo de la aplicación contiene las siguientes carpetas:

- **manifests**: contiene el archivo `AndroidManifest.xml`.
- **java**: contiene los archivos de código fuente de Java.
- **res**: Contiene todos los recursos, como diseños XML, cadenas de IU e imágenes de mapa de bits.

Ejemplo de una aplicación

En esa sección se muestra como crear una aplicación para Android muy simple y básica. El objetivo es exponer algunos conceptos que son requeridos al desarrollar una aplicación. Cabe recordar que no se pretende que este sea un curso intensivo, no sólo para esta sección sino para todo el documento escrito; simplemente se espera que quien lea esto conozca las herramientas y pueda ir de la nada a construir algo. Esto no es documentación oficial por lo que todo lo descrito, puede cambiar con las constantes actualizaciones de los entornos de desarrollo.

La aplicación está basada en la primera aplicación descrita en el libro *Android Programming The Big Nerd Ranch Guide* (Bill Phillips, Chris Stewart, Kristin Marsicano). En nuestro caso se llamará Data Structures Quiz. Esta aplicación prueba el conocimientos del usuario sobre estructuras de datos. El usuario presiona alguna de las opciones, para responder la pregunta que aparece e inmediatamente se refleja el resultado de si fue o no correcta la elección.

La aplicación consiste de una *actividad* y un *layout*.

Una *actividad* es una instancia de `Activity` una clase del Android SDK. Una actividad es la encargada de manejar la interacción del usuario con una pantalla de información.

Se escriben subclases de `Activity` para implementar la funcionalidad que la aplicación requiera. Una aplicación puede tener una o muchas subclases.

Data Structures Quiz es una aplicación muy simple, sólo necesitará una sola subclase de `Activity` llamada `QuizActivity`. `QuizActivity` manejará la interfaz de usuario.

Un *layout* define un conjunto de objetos en la IU y sus posiciones en la pantalla. Un layout está compuesto por definiciones escritas en un archivo XML. Cada definición es usada para crear un objeto que aparece en la pantalla, como un botón o texto.

Data Structures Quiz incluye un archivo del layout llamado `activity_main.xml`.

Creación del proyecto

El primer paso es crear un proyecto en Android Studio, El proyecto contiene los archivos para construir una aplicación.

Para crear un nuevo proyecto, simplemente se sigue el guía que provee Android Studio, basta con dar siguiente a todo por ahora. En resumen, se define el nombre de la aplicación, se especifica que dispositivos podrán ejecutar la aplicación, y se elige una actividad que servirá como punto de partida.

Diseño de la IU

El archivo `activity_main.xml`, debe contener los siguientes elementos:

- Un `LinearLayout` orientado verticalmente.
- Un `TextView`.
- Un `LinearLayout` orientado horizontalmente
- Dos botones (`Button`).

Modificando el archivo debe quedar como sigue:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context="com.example.ivan.datastructuresquiz.MainActivity"
    android:orientation="vertical">

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:padding="24dp"
        android:text="La búsqueda sobre un árbol binario de búsqueda, ¿qué
        ↪complejidad tiene? (Caso promedio)" />

    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        android:layout_gravity="center_horizontal">
```

```

<Button
    android:id="@+id/AButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="O(log(n))"
/>

<Button
    android:id="@+id/BButton"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="O(n) " />
</LinearLayout>
</LinearLayout>

```

Comparando el XML con la figura 2.11 cada widget tiene un elemento XML, y el nombre del elemento es el tipo de widget.

Cada elemento tiene un conjunto de atributos XML. Cada atributo es una instrucción sobre cómo debe configurarse el widget.

La jerarquía del view. Los widgets existen en una jerarquía de objetos **View** llamada la jerarquía view. La figura :: muestra la jerarquía que corresponde a nuestro archivo `activity_main.xml`.

La raíz de esta jerarquía es un **LinearLayout**. Como elemento raíz debe especificar el namespace XML de recursos de Android en `http://schemas.android.com/apk/res/android`.

LinearLayout hereda de una subclase de **View** llamada **ViewGroup**. Un **ViewGroup** es un widget que contiene un arreglo con otros widgets. Se utiliza **LinearLayout** cuando se desea que un conjunto de widgets estén acomodados sobre una sola columna o fila. Otras subclases **ViewGroup** son **FrameLayout**, **TabletLayout** y **RelativeLayout**.

Cuando un widget está contenido en un **ViewGroup**, se dice ser su hijo. La raíz **LinearLayout** tiene dos hijos: un **TextView** y otro **LinearLayout**. Éste último tiene dos botones hijos.

De diseños XML a objetos View Cuando se creó el proyecto, se generó una subclase de **Activity** llamada **MainActivity**. El archivo de esta clase está en el

directorio *app/java* del proyecto. La carpeta *java* es donde se encuentra el código de java de cada proyecto.

Mostremos el resultado final de cómo debe lucir la clase *MainActivity*, y después expliquemos cada parte.

```
public class MainActivity extends AppCompatActivity {
    /**
     * Añade atributos a la clase
     */
    private Button mAOptionButton;
    private Button mBOptionButton;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        /**
         * Obtiene las referencias a los widgets
         */
        mAOptionButton = (Button) findViewById(R.id.AButton);
        mBOptionButton = (Button) findViewById(R.id.BButton);

        /**
         * Configura los escuchadores
         */

        mAOptionButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                /**
                 * Muestra un Toast con el resultado para el usuario
                 */
                Toast.makeText(MainActivity.this, "Correct", Toast.LENGTH_SHORT).show();
            }
        });
        mBOptionButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                Toast.makeText(MainActivity.this, "Incorrect", Toast.LENGTH_SHORT).show();
            }
        });
    }
}
```

```
    }  
}
```

El método `onCreateBundle()` es llamado cuando una instancia de las subclase de `Activity` es creada. Cuando una actividad es creada, necesita de una IU para manejar. Para que esta actividad tenga su propia IU, se llama el método `setContentView`. Este método *infla* un layout y lo pone en la pantalla. Cuando un layout es inflado, cada uno de sus widgets es instanciado como lo definen sus atributos. Se especifica que layout inflar pasando el identificador del recurso.

Cada layout es un recurso. Un recurso es una pieza de la aplicación que no es código, cosas como imágenes, archivos XML, etc. Estos recursos están en el directorio `app/res`. Para acceder a un recurso desde el código se hace usando su identificador (*resource ID*).

Ahora, se definen dos variables miembro dentro de la clase, son de tipo `Button`, ya que se conectarán con los botones definidos en el layout.

En una actividad se pueden obtener las referencias a un widget inflado llamando al método `findViewById`. Este método recibe como parámetro el id del recurso y retorna un objeto `View`. En nuestro caso se asignan a las variables `mAOptionButton` y `mBOptionButton` los objetos `View`, a partir del id de los recursos de los botones.

Finalmente se crean unos escuchadores para cada botón. Las aplicaciones en Android son conducidas por eventos, por lo que cuando una aplicación espera por un evento en específico se dice que está «escuchando» por ese evento. La sintaxis es extraña pero simplemente se crea implementa una clase anónima y se sobreescriben sus métodos para manejar las respuestas a eventos. En nuestra aplicación los escuchadores de los botones muestran un mensaje en la pantalla cuando el usuario presiona alguno de los botones.

SDK para Android de NAOqi

Para ejecutar módulos de manera remota sobre los robots de NAOqi es necesario contar con un SDK para la plataforma y el lenguaje con el que se desee desarrollar. La plataforma en cuestión es Android, y Aldebaran brinda un SDK de Java específicamente para este sistema. Como ya se mencionó en la sección sobre **NAOqi**, el SDK de Java utiliza el **framework qi**.

El SDK provisto no ha sido actualizado y en el archivo `.jar` no es compatible con nuevas versiones de Android a partir de la 5.

El SDK se añade al proyecto como si fuera cualquier bibliotecas desarrollada por un tercero. Cambiando la vista del proyecto que viene por defecto, *Android*, a *Project*,



Figura 2.11: Capturas de la ejecución de la aplicación Data Structures Quiz.

se puede copiar el archivo `javanaoqi-sdk-2.1.4-android.jar` al directorio en la ubicación `app/libs`. Después se configura el archivo del gradle de la aplicación y del proyecto.

Configuración de Gradle

El sistema de construcción de Android Studio está basado en Gradle, y el plugin de Android para Gradle adiciona varias características que están contruidas específicamente para aplicaciones de Android.

Remark. *Gradle es un sistema de automatización de construcción de código abierto enfocado a la flexibilidad y desempeño.*

La última versión del plugin de Gradle compatible con el SDK que brinda Aldebaran es la 1.3.1. Por lo tanto después de sumar a nuestro directorio `app/libs` se configura lo siguiente en el archivo `buldgradle` del proyecto:

```
buildscript {
    dependencies {
        classpath 'com.android.tools.build:gradle:1.3.1'
    }
}
```

```
}
```

Por último sólo se agrega a las dependencias dentro del archivo `buildgradle` de la aplicación `compilefiles('libs/java-naoqi-sdk-2.1.4-android.jar')`, que compila el SDK de NAOqi.

Firebase para Android

Los servicios de Firebase utilizados son Firebase Realtime Database y Authentication. Este último a través de la biblioteca Firebase UI.

Antes de poder utilizar Firebase en una aplicación se debe agregar Firebase a ésta. Después de crear un proyecto en la consola de Firebase se siguen los pasos que ésta nos muestra para añadir Firebase a nuestras aplicaciones ya sea web o móviles, en nuestro caso para dispositivos Android. La consola genera un archivo de configuración que se añade a la aplicación y luego se agrega el SDK para poder utilizar las bibliotecas de Firebase en el proyecto.

Cada biblioteca de Firebase que se desee utilizar se añade a las dependencias. Por ejemplo, sumando a las dependencias de Gradle `com.google.firebaseio.firebaseio-database:11.6.2` se tiene acceso a la base de datos en tiempo real que provee Firebase.

Firebase Realtime Database

Para comenzar a usar la Firebase Realtime Database en la aplicación, se debe sumar a las dependencias del archivo `buildgradle`, de la aplicación, `compile'com.google.firebaseio.firebaseio-database:11.8.0'`.

Para leer y escribir en la base de datos, necesitas una instancia de `DatabaseReference`:

```
private DatabaseReference mDatabase;  
mDatabase = FirebaseDatabase.getInstance().getReference();
```

Leer y escribir datos. Para ejecutar una escritura básica se usa `setValue()` para guardar datos en una referencia que se especifique.

Los tipos de datos que se pueden almacenar son los siguientes:

- String
- Long

- Double
- Boolean
- Map<String, Object>
- List<Object>

También se puede pasar un objeto personalizado de Java con la restricción de que la definición de la clase debe tener un constructor predeterminado que no recibe parámetros y tiene métodos públicos para obtener a los atributos del objeto.

El siguiente fragmento muestra como escribir un nuevo usuario en la base de datos.

```
public void saveUserToDatabase(String uid, String email) {
    mDatabase.child("users/" + uid).setValue(email);
}
```

Detección de eventos en valores. Para leer datos en una ruta de acceso y escuchar actualizaciones, usa el método `addValueEventListener()` o el método `addListenerForSingleValueEvent()` para agregar un `ValueEventListener` a un objetos `DatabaseReference`.

Un escuchador `ValueEventListener` de una referencia de la base de datos tiene una método callback `onDataChange()` que obtiene una captura estática del contiene en la ruta determinada.

El ejemplo siguiente demuestra como un aplicación que muestra mensajes almacenados en la base de datos:

```
mMessagesReference.addValueEventListener(new ValueEventListener(){
    @Override
    public void onDataChange(DataSnapshot snapshot) {
        Message msg = snapshot.getValue(Message.class);
        //...
    }
})
```

El escuchador recibe un objeto `DataSnapshot` que contiene los datos de la ubicación específica al momentos del evento.

La diferencia entre `addValueEventListener()` y `addListenerForSingleValueEvent()` es que el primero se suscribe a cierta ubicación para escuchar cambios, y el segundo lee los datos de una ubicación una sola vez.

Actualización o borrado de datos. Para actualizar se llama al método `updateChildren()` y para eliminar datos en una referencia se usa `removeValue` o se pasa como parámetro un valor `null` a `setValue()`.

También existe el método `push()` para añadir una entrada con un identificador único sobre una referencia.

Firebase UI Authentication

Es una API para manejar el flujo del inicio de sesión con una dirección de correo electrónico, números de teléfono, y a través de proveedores como Google Sign-In, y Facebook Login. Está construido sobre Firebase Authentication.

Configuración. Como prerequisito, la aplicación debe estar configurada para utilizar Firebase. Después, se agrega a la biblioteca de FirebaseUI auth en las dependencias de `buildgradle` de la aplicación.

```
dependencies {
    compile 'com.firebaseio:firebase-ui-auth:3.0.0'
}
```

Uso de FirebaseUI para autenticación. Antes de llamar al flujo de autenticación de FirebaseUI, la aplicación debe checar que un usuario ya esté registrado de una sesión previa. Este caso es cuando el usuario inicio sesión y salió de la aplicación para luego regresar a esta.

```
 FirebaseAuth auth = FirebaseAuth.getInstance();
if (auth.getCurrentUser() != null) {
    // el usuario tiene abierta la sesión
} else {
    // el usuario no ha iniciado sesión
}
```

El punto de entrada para el flujo de la autenticación es la clase `com.firebaseio.ui.auth.AuthUI`.

Inicio de sesión. Si un usuario no ha iniciado sesión, entonces el proceso de inicio de sesión puede empezar creando un intent sign-in usando `AuthUI.SignInIntentBuilder`. Se puede recuperar una instancia del contructor llamando `createSignInIntentBuilder()` en la instancia retomada de `AuthUI`.

El constructor provee las siguientes opciones de personalización para el flujo de la autenticación:

- El conjunto de provedores de autenticación puede especificarse (Google, Twitter, Facebook)
- Un URL con los términos de servicio para la aplicación.
- Un tema personalizado puede ser especificado para el flujo, el cual se aplica a todas las actividades en el flujo para que haya consistencia de colores y tipografía.

Si no se requiere personalizar , y solo se necesita el correo electrónico para autenticarse, el flujo del inicio de sesión comienza como sigue:

```
// Un valor para el código de petición arbitrario
private static final int RC_SIGN_IN = 123;

// ...

startActivityForResult(
    // obtiene una instancia de AuthUI basado en la aplicación por defecto
    AuthUI.getInstance().createSignInIntentBuilder().build(),
    RC_SIGN_IN);
```

El segundo parámetro de `startActivityForResult()` (`RC_SIGN_IN`) es un código de petición para identificar la petición cuando el resultado retorna a la aplicación en el métodos `onActivityResult()`.

El flujo de la autenticación suministra varios códigos de respuesta, entre los más comunes están:

- `Activity.RESULT_OK`, si es el usuario inició sesión.
- `Activity.RESULT_CANCELED`, si el usuario canceló manualmente el inicio de sesión.
- `ErrorCodes.NO_NETWORK`, si no hay conexión a una red.
- `ErrorCodes.UNKNOWN_ERROR`, todos los otros errores.

Siguiendo con el ejemplo del inicio de sesión con el correo electrónico, el método `onActivityResult()` queda como sigue:

```
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    super.onActivityResult(requestCode, resultCode, data);
    // RC_SIGN_IN es el segundo parámetro que se pasó a
    startActivityForResult
```

```

if (requestCode == RC_SIGN_IN) {
    IdpResponse response = IdpResponse.fromResultIntent(data);

    // Si el usuario inició sesión exitosamente
    if (resultCode == RESULT_OK) {
        startActivity(SignedInActivity.createIntent(this, response));
        finish();
    } else {
        // El inicio de sesión falló
        if (response == null) {
            // El usuario presionó el botón de regresar
            showSnackbar(R.string.sign_in_cancelled);
            return;
        }

        if (response.getError().getErrorCode() == ErrorCodes.NO_
→NETWORK) {
            showSnackbar(R.string.no_internet_connection);
            return;
        }
        showSnackbar(R.string.unknown_error);
        Log.e(TAG, "Sign-in error: ", response.getError());
    }
}
}

```

Cierre de sesión. AuthUI brinda un método `signOut` simple que encapsula todo el proceso que conlleva un cierre de sesión. El método retorna una objeto Task que se marca completamente una vez que todo el proceso de cierre de sesión se ha completado.

```

public void signOut(){
    AuthUI.getInstance().signOut(this)
        .addOnCompleteListener(new OnCompleteListener<Void>() {
            @Override
            public void onComplete(@NonNull Task<Void> task) {
                if (task.isSuccessful()) {
                    startActivity(MyActivity.
→createIntent(SignInActivity.this));
                    finish();
                } else {
                    // Falló el cierre de sesión
                }
            }
        });
}

```

```
        }
    });
}
```

ButterKnife

ButterKnife es una biblioteca de viewbinding, esto quiere decir que genera objetos view a partir de recursos, pero lo que la hace especial es que evita el código repetitivo, como llamar a la función `findViewById`. ButterKnife ayuda a enlazar atributos, métodos y views.

Configuración. Para comenzar a utilizar butterknife en un proyecto. Solo se agrega a Las dependencias del `buildgradle` lo siguiente:

```
compile 'com.jakewharton:butterknife:8.8.1'
provided 'com.jakewharton:butterknife-compiler:8.8.1'
```

Uso. Para entender mejor las características que tiene ButterKnife supongamos que tenemos una actividad con un botón que reaccione a clicks. Sin usar ButterKnife la actividad queda como sigue:

```
public class MainActivity extends AppCompatActivity {
    private TextView mTextView;
    private Button mButton;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        mButton = (Button) findViewById(R.id.AButton);
        mTextView = (TextView) findViewById(R.id.texView);

        mAOptionButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                // Haz algo
            }
        });
}
```

```
    }  
}
```

Utilizando ButterKnife la actividad cambiaría a los siguiente:

```
public class MainActivity extends AppCompatActivity {  
    @BindView(R.id.textView)  
    TextView mTextView;  
  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
        // Enlaza butterknife a la actividad  
        ButterKnife.bind(this);  
    }  
  
    @OnClick(R.id.AButton)  
    public void OnClickMyButton() {  
        // Haz algo  
    }  
}
```

Al comparar se nota como evita duplicar código, y el código de los escuchadores es mucho más simple y parece explicarse por sí mismo.

Volley

Volley es una biblioteca HTTP que facilita el acceso a la red en aplicaciones de Android. Algunos de sus beneficios son los siguientes:

- Programación automática de peticiones a través de la red.
- Múltiples conexiones concurrentes.
- Soporte para priorizar peticiones.
- Una API para cancelar peticiones.
- Personalizable, por ejemplo, se puede añadir la opción de reintentar un petición.

Configuración. La manera más sencilla de añadir Volley a un proyecto es a través de la dependencia en el `build.gradle` de la aplicación.

```
dependencies {  
    ...  
    compile 'com.android.volley:volley:1.1.0'  
}
```

Enviando una petición simple. A un alto nivel, se utiliza Volley creando un objeto `RequestQueue` y pasándole objetos `Request`. `RequestQueue` administra hilos trabajadores para ejecutar operaciones en red, leyendo y escribiendo desde el caché, y parseando respuestas. Las peticiones hacen el parseo de respuestas en crudo y Volley se ocupa de remitir la respuesta parseada de regreso al hilo principal para su entrega.

Volley provee una método `Volley.newRequestQueue` que prepara una `RequestQueue`, usando valores por defecto, e inicia la cola. El siguiente ejemplo presenta cómo se crea una nueva cola de peticiones, así como una simple petición GET a `www.google.com`, recibiendo la respuesta en una cadena y mostrándola en un `TextView`.

```
@BindView(R.id.text)  
TextView mTextView;  
  
// Instancia a la cola de peticiones  
RequestQueue queue = Volley.newRequestQueue(this);  
String url ="http://www.google.com";  
  
// Solicita una respuesta en formato de cadena haciendo un GET al URL  
StringRequest stringRequest = new StringRequest(Request.Method.GET, url,  
        new Response.Listener<String>() {  
            @Override  
            public void onResponse(String response) {  
                mTextView.setText("Response is: "+ response.substring(0,500));  
            }  
        }, new Response.ErrorListener() {  
            @Override  
            public void onErrorResponse(VolleyError error) {  
                mTextView.setText("That didn't work!");  
            }  
});  
// Añade la petición a la cola RequestQueue  
queue.add(stringRequest);
```

Volley siempre entrega respuestas parseadas sobre el hilo principal. Ejecutarlo sobre el hilo principal es conveniente para poblar elementos de la interfaz de usuario con los datos recibidos.

Descripción de las clases principales

La aplicación está compuesta como cualquier otra, por una carpeta *manifests*, *res* y *java*. La carpeta *java* incluye los paquetes `com.lar.cloudnao.utilities` y `com.lar.cloudnao`.

Paquete utilities

Es un paquete compuesto por clases e interfaces que tienen como función realizar tareas asíncronas para manejar imágenes del robot, enviar peticiones a la API REST, manejar la IU, etcétera.

Interface OnRobotNameClickListener

Una interfaz con un método abstracto que sirve como callback cuando un usuario da click sobre alguno de los robots obtenidos desde Firebase y mostrados en un `RecyclerView`.

Interface VisionRequestCompleted

Interfaz con un método abstracto que se llama dentro de una tarea asíncrona para procesar un objeto que se obtiene de la respuesta a la solicitud del recurso `vision` de la API REST de CloudNAO. Hereda métodos de `AsyncTask`.

Class BatteryStatusTask

Esta clase obtiene el nivel de batería actual en el robot a través de la lectura del valor en `ALMemory`. Utiliza la clase `AsyncTask`. Muestra una `ImageView` que representa el estado de la batería y un `TextView` con el porcentaje de batería restante.

Class BoundingBoxesUtils

Una clase estática con un método para dibujar cuadros delimitadores sobre un `ImageView`. Para dibujar el cuadro delimitador recibe un arreglo con los valores necesarios para generar las coordenadas que se usan para dibujar.

Class ImageTask

Clase pública con métodos asíncronos para obtener una imagen desde la cámara del robot de manera remota. Obtiene un objeto que contiene la imagen, con el espacio de color por defecto del robot NAO (YUV) en un arreglo de bytes, convierte ese arreglo en una imagen jpeg, para después cargar ese objeto jpeg en un bitmap que sirve para llenar el `ImageView` que muestra la imagen enviada por el robot. Es una subclase de `AsyncTask`.

Class ItemsListAdapter

Esta clase es el `Adapter` para el `ListView` cuyos elementos están compuestos por un `CheckBox` y un `TextView`. Se utiliza al mostrar la lista de datos que se quieren guardar en Firebase desde `ALMemory`.

Class MyPagerAdapter

Subclase de `PagerAdapter` para el `ViewPager` del menú principal de la aplicación, por ahora tiene un tamaño máximo de 4.

Class ProxySubscriberItem

Clase pública para modelar los objetos de la lista con los datos de `ALMemory` que se guardarán en Firebase. `ItemsListAdapter` contiene un objeto `List` de instancias de `ProxySubscriberItem`. Los atributos son un valor booleano `mChecked` y una cadena `mProxyName` para saber si un ítem está marcado y el nombre de la llave en `ALMemory`, respectivamente.

Class RecyclerViewAdapter

La clase pública del `Adapter` para el `RecyclerView` que mostrará la lista de robots obtenidos desde Firebase y que le pertenecen al usuario que inció sesión en la aplicación.

Class RecyclerViewHolders

La clase que modela a los `ViewHolders` que representan cada robot obtenido de Firebase. Aquí se utiliza el método abstracto `onClickRobotName` como callback para ejecutar una acción al dar click sobre un elemento del `RecyclerView`.

Class RobotFromFirebase

La clase que modela los objetos de la lista de robots obtenidos desde Firebase, cada objeto contiene un identificador único y el nombre que el usuario asignó.

Class RobotLogs

Clase con métodos asíncronos para almacenar valores de **ALMemory** en Firebase, obtiene valores del robot y los almacena en una ubicación en Firebase con un **push()**, por lo que se genera un historial de logs.

Class ViewHolder

Clase que modela los **ViewHolders** de la lista de datos que se desean guardar en Firebase, están compuestos por un **CheckBox** y un **TextView** que describe a cada elemento.

Class VisionListElement

Clase pública que modela los objetos dentro de una lista que muestra resultados de las actividades de reconocimiento de rostros y objetos. Cada instancia tiene un identificador, que puede ser el nombre del sujeto reconocido , la confianza y las coordenadas de que delimitan al elemento encontrado (un cuadro que encierra un rostro, o un rectángulo que acota un objeto).

Class VisionRESTRequestsTask

La clase con métodos asíncronos para hacer peticiones a la API REST de Cloud-NAO. Recibe la imagen desde un **ImageView**, del que obtiene un bitmap y genera una cadena en base 64 que representa esa imagen. Después define la estructura y valores del JSON que se envía en una petición usando la biblioteca Volley. A partir de la respuesta obtenida ejecuta la función callback **onVisionRequestCompleted** para mostrar los resultados en el contexto de donde se solicitó.

Class VolleyResponseUtils

Clase estática para manejar la respuesta de **Volley**, que puede ser un error o puede ser la estructura que se espera.

Enum ModelObjectViewPager

Un enumerado de Java que representa todas las posibles páginas del ViewPager. Se utiliza en el menú de la aplicación. En este enumerado se almacenan el identificador del **Layout** de cada página y un título que le corresponde.

Actividades de la aplicación

Class FaceRecognitionActivity

La clase de la actividad que realiza el reconocimiento de rostros. Implementa la interfaz `VisionRequestCompleted` para llamar a su método abstracto desde la clase `VisionRESTRequestsTask`.

Class MenuActivity

La actividad que muestra el menú principal donde se elige entre otras actividades para interactuar con el robot. Muestra un `ViewPager` para navegar y elegir la actividad deseada.

Class ObjectDetectionActivity

La actividad encargada de la funcionalidad de la detección de objetos, implementa la interfaz `VisionRequestCompleted` para ejecutar su método `VisionRequestCompleted.onVisionRequestCompleted(Object)` como callback en `VisionRESTRequestsTask`.

Class OCRTranslationActivity

Clase de la actividad que se encarga del procesamiento de una imagen para detección de texto y su traducción. Implementa la interfaz `VisionRequestCompleted`.

Class RemoteControllerActivity

La actividad para el control remoto del robot. Además de poder controlar algunos movimientos como el caminado o el cambio de postura, se puede visualizar una imagen en vivo de la cámara del robot, así como permite enviar valores de `ALMemory` a una base de datos en la nube a través de Firebase.

Class Robot

No es una actividad pero es la clase pública principal para ejecutar módulos del framework de NAOqi en la aplicación. Es una interfaz para conectarse al robot y ejecutar algunos módulos de NAOqi. Usa el patrón de diseño Singleton para que solo exista una instancia del objeto y funcione como una variable global.

Class SelectRobotActivity

Actividad que muestra los robots registrados por el usuario así como la opción para añadir nuevos robots, o conectarse a uno de la lista. Implementa el método abstracto `OnRobotNameClickListener.onClickRobotName(String, String)`.

2.1.5. Firebase y el robot NAO

El robot NAO dentro de la arquitectura CloudNAO es sobre quien se aplican las otras tecnologías desarrolladas. La aplicación móvil, utilizando el SDK para Android de NAOqi, controla al robot, recibe imágenes para sus procesamiento en la nube, almacena logs para sus posteriores análisis usando la Firebase, también en la nube, y muchas funcionalidades más. Sin embargo, en esta sección se describe el robot como componente que no sólo recibe órdenes y envía datos sin ejecutar un programa localmente.

El cómputo en la nube como ya se habló en secciones pasadas, incluye también que el robot aunque no realice todo el procesamiento que conlleva el ejecutar un modelo de aprendizaje profundo para analizar imágenes, procese la salida de un algoritmo para realizar cierta tarea que desee.

En esta sección se describe cómo conectar al robot a la aplicación web utilizando como intermediario a Firebase.

Para conectar remotamente a la aplicación web con el robot NAO, es necesario un intermediario, ya que se quiere que la conexión no sea dentro de una misma red y la respuesta sea inmediata, Ese intermediario es Firebase, que sirve como backend para conectar estas plataformas.

El robot es una plataforma con recursos muy limitados, no hay un SDK específicamente hecho para el robot. Por lo anterior, se utiliza la API REST de Firebase Realtime Database para enviar y recibir información a través de internet. Sin embargo, a pesar de ser una API REST, fue necesaria la creación de un módulo escrito en Python que sirviera como una interfaz de alto nivel, donde hacer una petición HTTP fuera igual de sencillo que llamar una función en los SDK disponibles.

Módulo auxiliar para la API REST

Para utilizar la API REST de manera más simple se desarrolló una biblioteca que encapsulara los métodos de lectura, escritura y el streaming de datos.

Es un módulo de Python, compuesto por dos partes principales; la primera parte es aquella que maneja las peticiones de escritura y lectura usando los métodos HTTP antes mencionados, la segunda parte es la encargada de recibir las notificaciones de parte del servidor. Para la primera el uso de la biblioteca `requests` es suficiente.

Para la segunda, se emplean las bibliotecas `threading`, para que en un hilo se escuchen los cambios, `sseclient`, un cliente del protocolo **SSE**, y `socket`, el hilo del cliente **SSE**.

A continuación se describe cada componente del módulo `firebase`.

```
class firebase.ClosableSSEClient(*args, **kwargs)
```

Una clase pública desarrollada por el equipo de Firebase. Añade una funcionalidad al módulo `SSEClient` para abandonar correctamente el streaming.

```
class firebase.FirebaseDatabase(url)
```

Una clase para usar utilizar la API REST de Firebase con métodos de alto nivel. Sirve para crear algo similar a las *Referencias* denifidas por Firebase. Un objeto de esta clase representa una ubicación específica en la base de datos. Con los métodos se realizan operaciones de escritura y lectura sobre esa ubicación.

El constructor de la clase se encarga de definir la ubicación de la base de datos sobre la que se harán las operaciones. Necesita el diccionario `config` que contiene el valor del parámetro de autorización.

`add_event_listener(handler_function)`

Este método crea un hilo para escuchar los cambios en la ubicación actual de la base de datos, y manejar esos cambios en una función.

Parámetros `handler_function` – La función que maneja los eventos enviados por `firebase`.

`child(path)`

Se encarga de crear una nueva instancia con una ubicación relativa a la actual.

Parámetros `path` – La ruta que se añade a la ubicación actual.

Devuelve Un nuevo objeto de `FirebaseDatabase`

`get()`

Este método obtiene los datos asociados con la ubicación del objeto a través del método `GET` del protocolo HTTP.

`push(data)`

Genera una nueva ubicación hija de la actual. Esta ubicación tiene un llave aleatoria única y se escriben los datos en esta. Por ejemplo, al hacer un `push({ "name" : "Rick Deckard" })` a `/users/` se produce `/users/-L43aJQFQNzAIxry8_6g/name/` con un valor `Rick Deckard`. Simplemente hace un

POST con los datos a enviar a la URL del objeto. Recibe como parámetro cualquier estructura válida en un JSON.

`remove()`

Elimina todos los datos en la ubicación actual.

`set(data)`

Escribe datos en la ubicación actual. Esto sobreescribe cualquier información que contenga. Hace un PUT sobre la ubicación actual. :param *data*: Los datos que se envían en la petición. :type *data*: Cualquier tipo de datos válido en un JSON.

`update(data)`

Escribe multiples valores en la ubicación actual de la base de datos. A diferencia de `set()` solo actualiza los valores que se desean de acuerdo a la ubicación actual. Se hace un PATCH al URL.

```
class firebase.RemoteThread(url, handler)
```

Una clase creada por el equipo de Firebase para que un hilo se encargue de los eventos enviados por Firebase, con una funcionalidad que añadí para que una función callback maneje los datos enviados por Firebase.

Módulo para utilizar la API de NAOqi

Este módulo es el que se ocupa de integrar la API de NAOqi en la aplicación que se corre sobre el robot. Aquí se crean los proxies a los módulos de NAOqi, y los métodos que se utilizan para obtener datos de la memoria del robot, realizar un movimiento, hablar, entre otros.

```
class nao_robot.Robot(ip_address, port)
```

Una clase para encapsular todos los métodos del robot que van a recibir o enviar información a Firebase. Los atributos de clase y sus valores son los siguientes:

```
cam_idx = 0
resolution = 0
color_space = 11
fps = 10
camera_subscriber = 'cameraSubscriber0001'
sonar_subscriber = 'sonarSubscriber0001'
move_config = [
    ['MaxStepFrequency', 0.5],
```

```

['MaxStepTheta', 0.349],
['MaxStepX', 0.04],
['MaxStepY', 0.14],
['StepHeight', 0.02]
]

```

El constructor de la clase crea los proxies a los módulos de NAOqi. Recibe la dirección ip y el puerto del robot. A continuación se describen los métodos de esta clase.

change_posture(*posture*)

Cambia entre dos posturas del robot, **Stand** y **Crouch**.

get_battery_level()

Solicita el valor del nivel de la batería desde la memoria del robot y lo retorna.

get_image_from_robot()

Obtiene la imagen del robot codificada en base64. Primero con **getImageRemote()** recibe el contenedor de la imagen, y luego, el arreglo de bytes de la imagen en el contenedor, se guarda en una estructura de PIL para que pueda convertirse en una cadena.

get_values_from_memory()

Obtiene valores de la memoria del robot y retorna un diccionario con sus llaves y valores.

move_joint(*joint_name, angle*)

Mueve una articulación del robot con respecto a la posición inicial de una articulación. Primero activa la rigidez del motor que mueve la articulación, realiza el movimiento en dos segundos y luego desactiva la rigidez del motor.

move_robot(_x, _y, _theta)

Método para ejecutar hacer caminar utilizando **moveToward** de la API de NAOqi.

Parámetros

- **_x** – La velocidad (-1, 1) sobre el eje **X**
- **_y** – La velocidad (-1, 1) sobre el eje **Y**
- **_theta** – La velocidad (-1, 1) sobre el eje **Z**

say_speech(*text*)

Ejecuta la función **say** de **ALTextToSpeech**. Dice la cadena enviada como parámetro.

set_move_config(move_config_map)

Cambia los parámetros de caminado del robot. Recibe un diccionario con los nuevos parámetros y sus valores. Los procesa para ver si son válidos y cambiarlos.

stop_movement()

Método para detener los movimientos del robot, llama a `stopMove()` de `ALMotion`.

subscribe_to_camera()

Se suscribe a la cámara del robot, con los parámetros que se definieron en los atributos de la clase.

subscribe_to_sonar()

Se suscribe al sonar para que se actualicen los valores en la memoria con las distancias a obstáculos.

unsubscribe_to_camera()

Da de baja al suscriptor de la cámara

unsubscribe_to_sonar()

Se apagan los sonares.

Conexión de la API REST de Firebase y NAOqi

Después de contar con una biblioteca cliente para realizar peticiones a la API REST de Firebase y con un módulo que utilice la API de NAOqi sólo resta unirlos en una aplicación que sea fácil de ejecutar y que responda con funciones de módulos de NAOqi a actualizaciones enviadas por Firebase o que envíe información a ubicaciones de la base de datos.

Este aplicación debe poder ejecutarse local y remotamente (dentro de la misma red) en el robot.

Para mantener simple la ejecución del script que conecta al robot con Firebase la aplicación está dividida en cinco módulos. Estos incluyen a los módulos descritos anteriormente, `firebase` y `nao_robot`.

app.py

app.main()

Inicia la aplicación, obtiene la dirección ip y puerto del robot a través de los argumentos enviados en la línea de comandos, luego llama a la función `run()` del módulo `fire_nao`.

config.py

Este archivo únicamente tiene un diccionario con la configuración para Firebase con la siguiente estructura:

```
config = {  
    "databaseURL": "https://url-de-firebase.com/",  
    "auth": "API KEY del proyecto de Firebase",  
    "robotUID": "identificador único del robot generado por Firebase"  
}
```

fire_nao.py

Este módulo es quien integra los módulos `firebase` y para comunicar al robot con la aplicación web a través de Firebase.

2.1.6. Aplicación web de CloudNAO

La aplicación web para la arquitectura CloudNAO es un componente que permite a usuarios interactuar con el robot NAO, enviando comandos para que éste ejecute algunos movimientos, diga algo, o actualice ciertos parámetros en su memoria; y además reciba valores de la memoria del robot para generar un historial de información del robot o simplemente recibir una imagen de lo que el robot visualiza a través de su cámara. Esta aplicación forma parte de lo que llamamos *Robot logs* y *Robot Remote Control*.

A pesar de que el objetivo principal de esta aplicación web es ser el front-end de la conexión entre el robot NAO directamente con la Firebase Realtime Database, la unión de estos tres elementos pretende mostrar un caso de uso del cómputo en la nube, que no es el procesamiento de imágenes o la ejecución de un algoritmo de aprendizaje automático que el robot difícilmente podría ejecutar con los recursos que cuenta. En cambio, que Firebase con su Realtime Database, Hosting y Authentication, permiten generar grandes volúmenes de datos globalmente disponibles, y protegidos.

Como se mencionó, la aplicación es una herramienta que simplemente recibe y envía información de la Realtime Database de Firebase todo a través de una interfaz gráfica sencilla para los usuarios del robot NAO.

Además de depender del SDK para Web de Firebase, otras dos dependencias están detrás de la aplicación, React, una biblioteca de Javascript para construir interfaces de usuario y Semantic UI.,

La aplicación tiene una estructura muy sencilla, cumple con los elementos básicos de una aplicación web moderna. Un login de usuario, registro de uno nuevo, y un tablero donde el usuario interactúe con el robot.

En las siguientes secciones se describen a detalle los elementos que confoman este componente, Firebase y React como herramientas de desarrollo, una descripción más detallada de la aplicación y la guía para los desarrolladores que mantienen la aplicación.

Descripción de la aplicación

La aplicación es bastante simple e intuitiva, el flujo que se sigue para utilizarla es el siguiente:

1. El usuario inicia sesión o se registra si no ha creado una cuenta.
2. Es redireccionado a un panel para interactuar con el robot, después de haber seleccionado alguno de una lista o de haber creado uno nuevo.
3. Puede cerrar la sesión o esta se mantiene activa.

Como se puede ver, la aplicación web cuenta con algunas características muy similares a la aplicación móvil. Cuenta con las funcionalidades para el registro de usuarios, que cada usuario pueda utilizar varios robots e interactuar con ellos usando un control remoto, o recibiendo valores enviados por el mimos Robot a Firebase.

Funcionalidades

Inicio de sesión. Un usuario puede iniciar sesión si ya se ha registrado, sino tiene la opción de crear una nueva cuenta. El usuario no necesariamente debe crearse dentro de la aplicación web, si se ha registrado usando la aplicación móvil, entonces puede usar el mismo nombre de usuario y contraseña.

Panel de interacción con el robot. La principal función de esta aplicación es conectarse remóticamente a un robot NAO usando como intermediario a Firebase. Si se desea enviar una orden al robot primero se escribe esa orden en una ubicación en la base de datos, el robot recibe dicha actualización y la procesa para ejecutar tal comando.

El panel cuenta la estructura mostrada en la figura Figura 2.13:

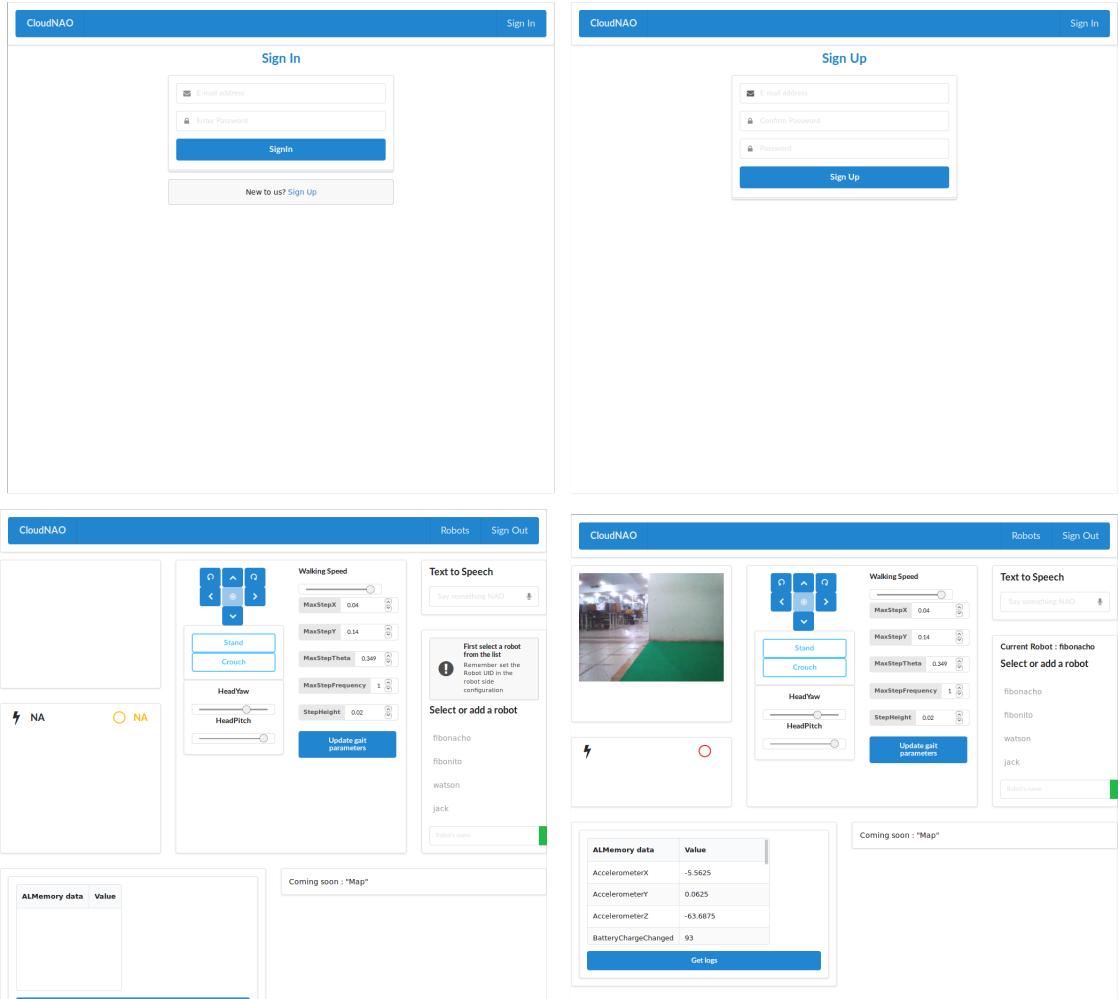


Figura 2.12: El usuario inicia sesión o se registra, es redireccionado a un panel, el usuario debe elegir un robot de la lista para poder interactuar con él.

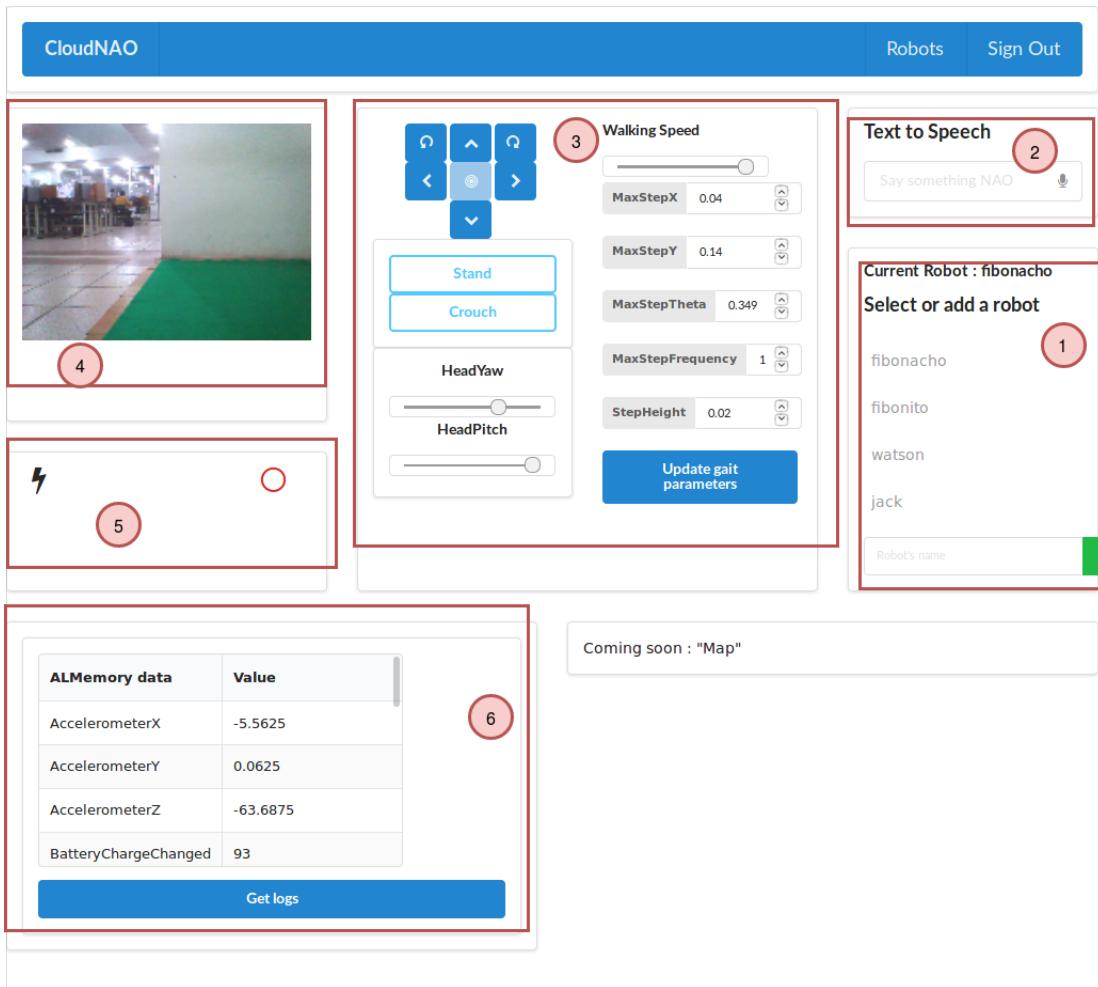


Figura 2.13: Estructura del panel dentro de la aplicación

1. Lista de los robots que ha añadido el usuario, aquí también puede agregar uno. Primero hay que seleccionar un robot de la lista para poder ver sus logs, o la última imagen que envío de sus cámara, etc.
2. Aquí el usuario puede escribir un texto breve para que el robot lo repita oralmente.
3. Control remoto del robot para realizar algunos movimientos como caminar, cambiar entre dos posturas y mover las articulaciones de la cabeza. Se puede cambiar la velocidad con la que camina y otros parámetros de caminado definidos por la API de NAOqi.
4. Muestra una imagen tomada con la cámara superior del robot, funciona para obtener una imágen en vivo enviada por el robot cuando éste está conectado a Firebase.
5. Aquí aparece el estado de la conexión con el robot, si el robot ha ejecutado el programa que se conecta a Firebase, se muestra un estado en línea y el nivel de la batería.
6. Una tabla con valores de ALMemory enviados por el robot. También permite descargar el historial completo de logs.

Cierre de sesión. El usuario puede cerrar su sesión o esta se mantiene activa en el navegador hasta que se deseé.

Cabe señalar nuevamente que para usar todas las funcionalidades de esta aplicación se debe ejecutar el programa del lado del robot que se conecta a Firebase para recibir y enviar información, si esto no se lleva a cabo, sólo se muestran los últimos datos enviados por el robot o por la aplicación móvil.

Cada robot que registra un usuario tiene un identificador único, este identificador le sirve para escribir en las ubicaciones den la base de datos que le corresponden. Al pasar el puntero del ratón sobre un robot en la lista de robots, se muestra su identificador único que es el que se añade al archivo de configuración del programa que se corre sobre el robot.

Guía para desarrolladores

En la aplicación se implementa únicamente la parte del front end, de la parte del back end, se encarga Firebase. Por esto, las únicas herramientas que se usaron fueron React y el SDK de Firebase para la autenticación de usuarios y la base de datos.

React

React es una biblioteca de Javascript para crear interfaces de usuario lanzada por Facebook en 2013. Tiene tres características que la definen y distinguen de otras biblioteca o frameworks y son:

- Es **declarativa**, ya que permite crear interfaces de manera sencilla. Sólo se diseña una vista para cada estado de la aplicación; React actualizará de manera eficiente y **rederizará** los componentes correctos cuando los datos cambian.
- Está **basado en componentes**, construye componentes encapsulados que manejan su propio estado, después se integran con otros para hacer IU más complejas.
- React puede *renderizar* sobre un servidor usando node o en dispositivos móviles con React Native.

Los *componentes* dividen la IU en piezas independientes y reutilizables; además permite pensar en cada una de manera aislada. Conceptualmente, los componentes son como funciones de Javascript. Aceptan entradas arbitrarias llamadas **props**, abreviación de propiedades, y regresan elementos de React describiendo lo que debe aparecer en la pantalla.

Los *elementos* son los bloques más básicos en las aplicaciones de React. Un elemento describe todo que se quiere visualizar en la pantalla.

```
const element = <h1>Hola mundo</h1>
```

El método **render** regresa una descripción de que es lo que se quiere ver en la pantalla. React toma esa descripción y muestra el resultado. En particular, **render** retorna un elemento de React, que es una descripción ligera de lo que se quiere renderizar.

Para el desarrollo de una aplicación en React se utiliza una sintaxis especial llamada *JSX* (JavaScript XML), que no es requerida, pero al ser una extensión de JavaScript que permite utilizar un código parecido al HTML con todo el poder de JavaScript se cumple el objetivo de React de no separar la lógica de renderizado y la de la interfaz de usuario.

Ejemplo de una aplicación. La manera más simple para crear una aplicación en React es através del kit inicial sin configuraciones para React lanzado por Facebook llamada `create-react-app`.

Para iniciar con esta herramienta primero se instala a través del manejador de paquetes de node, `npm`:

```
npm install -g create-react-app
```

Ahora se puede crear y lanzar una aplicación con React.

```
create-react-app ejemplo_react_app  
cd react_ejemplo_app
```

La estructura del directorio creado es la siguiente:

```
ejemplo_react_app/  
    ├── node_modules  
    ├── package.json  
    ├── package-lock.json  
    ├── public  
    │   ├── favicon.ico  
    │   ├── index.html  
    │   └── manifest.json  
    ├── README.md  
    └── src  
        ├── App.css  
        ├── App.js  
        ├── App.test.js  
        ├── index.css  
        ├── index.js  
        ├── logo.svg  
        └── registerServiceWorker.js
```

- **node_modules/**. Contiene todos los paquetes de node que fueron instalados vía npm. Como se usó create-react-app, algunos módulos ya fueron instalados.
- **package.json**. El archivo que muestra las lista de dependencias y otras opciones de configuración del proyecto.
- **.gitignore**. Este archivo indica todos los archivos y carpetas que nos deben añadirse a un repositorio remoto cuando se usa git.
- **public/**. Esta carpeta guarda todos los archivos cuando se despliega un proyecto en modo de producción.

En un principio todo lo que se necesita está localizado en la carpeta *src/*. El enfoque principal está dirigido al archivo *src/App.js* para implementar los componentes de React. Este se usa para implementar la aplicación, pero con el crecimiento de un proyecto siempre es necesario dividir los componentes en múltiples archivos, donde cada archivo mantiene uno o algunos componentes por su cuenta.

La aplicación generada con *create-react-app* es un proyecto de npm. Se puede usar

npm para instalar o eliminar paquetes de node. Además viene con los siguientes scripts de npm para la línea de comandos.

```
# Ejecuta la aplicación en http://localhost:3000
npm start

# Ejecutar los tests
npm test

# Construye la aplicación para producción
npm run build
```

Esos scripts están definidos en el *package.json*.

El siguiente fragmento de código muestra una subclase de `React.Component` llamada `PodcastsList`.

```
class PodcastsList extends React.Component {
  render() {
    return (
      <div className="shopping-list">
        <h1>Podcasts for {this.props.name}</h1>
        <ul>
          <li>Simulacro</li>
          <li>Olallo Rubio</li>
          <li>Revolver</li>
        </ul>
      </div>
    );
  }
}
```

Una forma de añadir la clase anterior a nuestro proyecto en `create-react-app` es modificando el archivo `index.js` dentro de `src/` cambiando `<App>` por el `<PodcastsList name="Donkey" />`, si la aplicación se está ejecutando solo se mostrará el mensaje y ningún otro elemento de los generados por `create-react-app`.

Figura 2.14: Resultado de la aplicación en un navegador.

Podcasts for Donkey

- Simulacro
- Olallo Rubio
- Revolver

Firebase para aplicaciones web Para utilizar los servicios Firebase en una aplicación web se siguen los mismos pasos que para el caso de aplicaciones para Android. Desde la consola de Firebase se crea el proyecto y se obtiene un fragmento de código de inicialización. Ese código contiene información de inicialización para configurar el SDK de Firebase JavaScript a fin de usar Authentication, Cloud Storage, Realtime Database y Cloud Firestore.

Se puede acceder a cada servicio desde el espacio de nombres de `firebase`:

- `firebase.auth()` - Authentication
- `firebase.storage()` - Cloud Storage
- `firebase.database()` - Realtime Database
- `firebase.firestore()` - Cloud Firestore

El proceso para usar el servicio de Realtime Database y Authentication es el mismo que en Android. Los métodos de lectura y escritura se mantienen cambiando únicamente la sintaxis.

Firebase Realtime Database Para leer la base de datos o escribir en ella, se necesita una instancia de `firebase.database.Reference`:

```
// Obtiene una referencia al servicio de la base de datos
var database = firebase.database();
```

Lectura y escritura de datos. Para recuperar los datos de Firebase, se debe agregar un escuchador asíncrono a `firebase.database.Reference`. El escuchador se activa una vez para el estado inicial de los datos y otra vez cuando los datos cambian.

Para ejecutar operaciones de escritura básicas, puedes usar `set()` para guardar datos en una referencia que especifiques y reemplazar los datos existentes en esa ruta de acceso. Por ejemplo si se desea añadir un usuario a la base de datos, una opción es la siguiente:

```
function writeUserData(userId, name, email, imageUrl) {
  firebase.database().ref('users/' + userId).set({
    username: name,
    email: email,
    profile_picture : imageUrl
  });
}
```

`set()` sobrescribe los datos en la ubicación que se especifica, incluidos los nodos secundarios.

Detección de eventos en valores. Si deseas leer datos de una ruta de acceso y escuchar para detectar cambios, usa los métodos `on()` o `once()` de `firebase.database.Reference` para observar eventos.

El evento más común es `value` y permite leer una instantánea estática del contenido de una ruta de acceso determinada, en el estado en que se encontraba en el momento del evento. Este método se activa cuando se vincula el escuchador y se vuelve a activar cada vez que cambian los datos (incluidos los de nivel secundario). La devolución de llamada del evento recibe una instantánea que contiene todos los datos de dicha ubicación, incluidos los datos secundarios. Si no hay datos, la instantánea tiene un valor nulo.

En el siguiente ejemplo se muestra una aplicación donde se recupera el valor de un recurso en la base de datos:

```
var resourceRef = firebase.database().ref('resource');
resourceRef.on('value', function(snapshot) {
  showValueOnScreen(snapshot.val());
});
```

El escuchador recibe una `snapshot` que contiene los datos de la ubicación específica en la base de datos en el momento del evento. Se pueden recuperar los datos de la `snapshot` con el método `val()`.

Actualizar o borrar datos Para escribir de forma simultánea en elementos secundarios específicos de un nodo sin sobrescribir otros nodos secundarios, usa el método `update()`.

También existe el método `push()` para añadir una entrada con un identificador único sobre una referencia.

La forma más sencilla de borrar datos es llamar a `remove()` en una referencia a la ubicación de los datos. Para borrar, también puedes especificar nulo como el valor de otra operación de escritura, como `set()` o `update()`.

Firebase Authentication

De este servicio se utilizó la autenticación con correo electrónico y contraseña. El SDK de Firebase Authentication proporciona métodos para crear y administrar usuarios que utilizan sus direcciones de correo electrónico y sus contraseñas para acceder.

Para crear una cuenta de usuario nueva se pasa la dirección de correo electrónico y la contraseña del nuevo usuario a `createUserWithEmailAndPassword` para crear la cuenta nueva.

```
firebase.auth().createUserWithEmailAndPassword(email, password).  
  ↪catch(function(error) {  
    // El manejo de errores va aquí  
    var errorCode = error.code;  
    var errorMessage = error.message;  
  });
```

Cuando un usuario accede a tus apps, se pasan la dirección de correo electrónico y la contraseña al método `signInWithEmailAndPassword`.

```
firebase.auth().signInWithEmailAndPassword(email, password).  
  ↪catch(function(error) {  
    // Aquí se manejan los errores  
    var errorCode = error.code;  
    var errorMessage = error.message;  
    // ...  
  });
```

La manera recomendada de obtener el usuario actual es establecer un observador en el objeto Auth:

```
firebase.auth().onAuthStateChanged(function(user) {  
  if (user) {  
    // El usuario ha iniciado sesión  
  } else {  
    // No hay un usuario activo  
  }  
});
```

También se puede usar la propiedad `currentUser` para obtener el usuario que accedió. Si un usuario no accedió a su cuenta, `currentUser` mostrará un resultado nulo:

```
var user = firebase.auth().currentUser;  
if (user) {  
  // El usuario ha iniciado sesión  
} else {  
  // No hay un usuario activo  
}
```

Para obtener la información del perfil de un usuario, puedes usar las propiedades

de una instancia de User. Por ejemplo:

```
var user = firebase.auth().currentUser;
var name, email, photoUrl, uid, emailVerified;

if (user != null) {
  name = user.displayName;
  email = user.email;
  photoUrl = user.photoURL;
  emailVerified = user.emailVerified;
  uid = user.uid; // El id del usuario es único en un proyecto de Firebase
}

}
```

Para salir de la sesión de un usuario, se llama a `signOut`:

```
firebase.auth().signOut().then(function() {
  // Se cierra la sesión exitosamente
}).catch(function(error) {
  // Un error ocurrió
});
```

Firebase Hosting Este servicio permite implementar y alojar los recursos estáticos de una aplicación fácilmente (HTML, CSS, JavaScript, etc.). Todo el contenido se transmite a través de HTTPS y está respaldado por una CDN global.

La ruta de implementación se muestra en las siguientes subsecciones.

Instalar Firebase CLI. La CLI (interfaz de línea de comandos) de Firebase necesita Node.js y npm.

Para instalar Firebase CLI a través de npm se escribe los siguiente:

```
npm install -g firebase-tools
```

Esto instala el comando `firebase` de manera global.

Iniciar la aplicación. Si se ejecuta el comando `firebase init`, se crea un archivo de configuración `firebase.json` en la raíz del directorio del proyecto.

Agregar un archivo. Cuando se inicializa una aplicación, se pedirá proporcionar un directorio para usarlo como raíz pública (el valor predeterminado es «public»). Si aún no se tiene un archivo index.html válido en tu directorio raíz público, se creará uno.

Lanzar un sitio web. Ejecutar `firebase deploy` implementará la aplicación en el dominio <APP-DE-FIREBASE>.firebaseapp.com

Descripción de los componentes

La aplicación se creó utilizando la herramienta `create-react-app` y después se le agregaron todos los servicios de Firebase que se ocuparon.

A partir de la estructura generada por `create-react-app` todo el código fuente se agregó a la carpeta `src`. Éste tiene la siguiente estructura:

```
src/
  ├── App.js
  ├── App.test.js
  ├── auth.js
  └── components
      ├── dashboard.js
      ├── error_msg.js
      ├── landing_page.js
      ├── live_image_from_robot.js
      ├── log_table.js
      ├── navigation_bar.js
      ├── not_found.js
      ├── robot_commands.js
      ├── robot_log.js
      ├── robot_selection.js
      ├── robot_status.js
      ├── signin.js
      ├── signout.js
      ├── signup.js
      ├── text_to_speech.js
      ├── with_authentication.js
      └── with_authorization.js
  ├── constants
  │   └── routes.js
  ├── fire.js
  └── index.css
```

```
└── index.js  
└── registerServiceWorker.js
```

App.js Se define el componente principal de la aplicación App, que se renderizará al iniciarla. Se define el elemento Router de React-Router que define las rutas de los componentes que se renderizarán de acuerdo a lo elegido por el usuario.

auth.js Aquí se definen las funciones para el inicio de sesión, registro de un usuario y el cierre de sesión.

fire.js Aquí está la configuración de Firebase para usar los servicios de la base de datos y de la autenticación.

index.js En este archivo se llama al método render de ReactDOM para renderizar al componente App.

Dentro del directorio components están los archivos con los componentes de React para la aplicación.

dashboard.js Archivo con el componente Dashboard que muestra el panel de interacción con el robot. Como estados define al usuario actual, un id y el nombre de un robot, el robot que se selecciona en la lista de robots.

En el método render() incluye otros componentes: LiveImage, RobotStatus, Robot Commands, TextToSpeech, RobotList y RobotLogs.

error_msg.js Exporta al componente ErrorMessage, un mensaje de error utilizado en el SignUp y SignIn del usuario, pero puede adaptarse a cualquier contexto pues simplemente recibe el texto del error generado.

landing_page.js Incluye a la subclase de React.Component Landing que se muestra al abrir la aplicación, estando o no una sesión activa.

live_image_from_robot.js Define al componente LiveImage que muestra la última imagen enviada por la cámara del robot. Tiene una imagen por defecto, y espera una imagen codificada en base 64 obtenida desde Firebase.

log_table.js Se define TableFromObject, un componente que genera una tabla a partir de un JSON. Dos columnas lo componen, en la primera están las llaves y en la segunda los valores de los atributos del objeto de javascript. Se utiliza para mostrar los logs adquiridos de Firebase.

navigation_bar.js Hay tres componentes dentro de este archivo: `NavigationAut`, el menú de navegación cuando el usuario no ha iniciado sesión, `NavigationNonAuth`, el menú cuando ha iniciado sesión y `NavigationBar`, el que muestra al menú que corresponde al estado del usuario.

robot_commands.js Exporta el componente `RobotCommands` que es el control remoto del robot, aquí se envía información a la de Firebase para que el robot realice acciones como caminar, cambiar de postura, o mover la cabeza, también se pueden actualizar algunos parámetros de caminado.

robot_log.js Define a la clase `RobotLogs`, subclase de `React.Component`, que renderiza la tabla de logs con la información que se obtiene de Firebase, también permite descargar el historial completo de logs para cada robot.

robot_selection.js Se define al componente `RobotList` el cual maneja la selección de un robot y actualiza al `Dashboard` para que envíe a sus hijos el id del robot seleccionado.

robot_status.js

Contiene al componente `RobotStatus` que muestra la batería y el estado de la conexión con el robot.

signin.js Se encuentra la clase `SignIn`, el componente que maneja el inicio de sesión de un usuario, utiliza al objeto `auth` para que el usuario utilice su dirección de correo y una contraseña.

signout.js Define al componente `SignOutMenuItem` encargado de cerrar la sesión del usuario.

signup.js Contiene al componente `SignUp` en el que un nuevo usuario se registra. Utiliza las funciones de autenticación de Firebase y además almacena al nuevo usuario a la base de datos.

text_to_speech.js Define la clase `TextToSpeech`, el componente que envía una cadena de texto a Firebase para que el robot la replique oralmente.

constants/routes.js Este archivo contiene las rutas de la aplicación.

```
export const SIGN_IN = '/signin';
export const SIGN_UP = '/signup';
export const LANDING = '/';
export const ROBOTS = '/robots';
```

La aplicación inicia mostrando el componente de la ruta `LANDING`, que es `Landing`, el usuario selecciona `Sign In` en el menú de navegación se dirige a la ruta

SIGN_IN, si es un nuevo usuario puede dar click en el botón que envía a la ruta SIGN_UP. Después de realizar un inicio de sesión de cualquiera de las dos formas anteriores se redirecciona a la ruta ROBOTS que muestra al componente Dashboard.

2.2. Implementación de casos de estudio con servicios en la nube existente

2.2.1. El robot NAO como asistente de voz

Capítulo 3

Un servicio de cómputo en la nube sobre la arquitectura CloudNAO

Para completar la arquitectura CloudNAO falta implementar un modelo de aprendizaje profundo que se brinde como un servicio web para ser consumido por robots NAO. A pesar de que el servicio de detección de objetos es mantenido por el LAR, y se adaptó para ser consumido a través de la API REST, no fue construido desde cero ya que es parte de la API de detección de objetos de Tensorflow. Es por eso que como parte final del proyecto se desarrolló un modelo para que resuelva la tarea de clasificar imágenes.

El problema de clasificación de imágenes es la tarea de asignarle a una imagen de entrada una etiqueta a partir de un conjunto de categorías. Este es uno de los principales problemas dentro del campo de la visión computacional, que a pesar de su simplicidad tiene bastantes aplicaciones prácticas. Entre esas aplicaciones, muchas interesan al campo de la robótica móvil, por ejemplo, para la navegación de un robot de manera autónoma, nos gustaría que supiera en qué lugar está simplemente con una fotografía que obtenga en ese momento desde sus cámaras, así podría saber si ha llegado al lugar de su objetivo, o a partir de la zona donde se ubica planear una trayectoria.

Lo anterior nos inspiró en la creación de un modelo que clasificara imágenes de algunos lugares sobre los que podría navegar el robot NAO. Como solución a este problema de clasificación se propuso usar una red neuronal convolucional, que recibiera como entrada un arreglo con los píxeles de una imagen tomada por el robot, y la salida fuera la categoría a la que pertenece esa imagen. Las clases en las que se desea clasificar las imágenes son lugares alrededor del Laboratorio de Algoritmos para la Robótica, que se ubica en el cubículo 15 del Centro de Desarrollo Tecnológico de la FES Acatlán. Se eligieron las siguientes cuatro zonas:

- El cubículo.
- La salida de emergencia.
- La cancha de entrenamiento de fútbol para el robot NAO.
- Zona de trabajo del Laboratorio.

En este capítulo se describe con detalle la confección del conjunto de datos, la arquitectura de la red convolucional, la implementación del modelo utilizando Python y TensorFlow, los resultados obtenidos y finalmente cómo se integra este modelo dentro de la API de CloudNAO

3.1. Descripción del servicio sobre CloudNAO

3.1.1. Descripción del conjunto de datos

Son cuatro las categorías en las que se quiere clasificar una imagen obtenida por el robot: el cubículo, la salida de emergencia, la cancha de fútbol y la zona de trabajo.



Figura 3.1: Fotografías de las cuatro zonas dentro del laboratorio. De izquierda a derecha y de arriba a abajo: salida, cancha, zona de trabajo y cubículo.

Las imágenes se obtuvieron desde el robot usando el módulo de `ALVideoRecorder`, el cual permite guardar secuencias de video utilizando las cámaras del robot y guardarlas en su disco. De los videos se extrajeron algunos fotogramas, no todos para evitar redundancias, los cuales se separaron en carpetas de acuerdo al lugar (categoría) que pertenecían. Después de este proceso de obtención de imágenes a partir de video y de su separación en cada clase, se generaron dos conjuntos, el de entrenamiento y el de prueba. El conjunto de entrenamiento está compuesto por

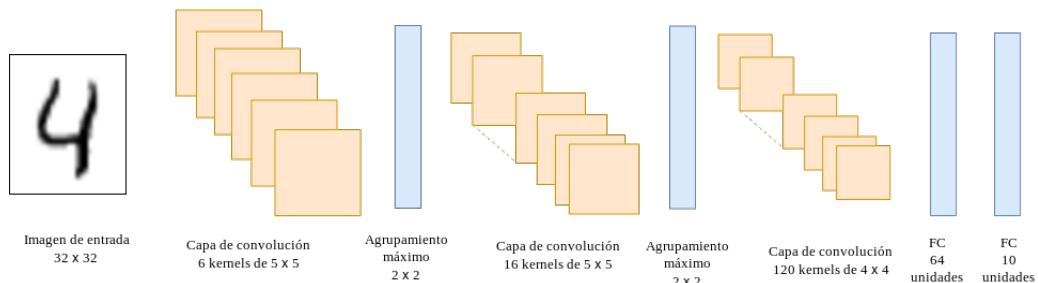
6000 imágenes, 1500 por cada clase. El conjunto de prueba contiene 2112 imágenes, 528 por cada categoría. En ambos conjuntos cada imagen es de 60×80 pixeles y de tres canales por ser imágenes en el modelo de color RGB.

3.1.2. Arquitectura de la CNN

La selección de una arquitectura para la CNN y de sus parámetros se realizó a través de pruebas. Se tomó como base la arquitectura LeNet-5 (una variante de la arquitectura LeNet que fue una de las primeras CNN que operaba sobre el conjunto MNIST de dígitos escritos a mano) y se probó cambiando diversos parámetros como el número y tamaño de los kernels, la tasa de aprendizaje y el número de unidades en las capas completamente conectadas.

El primer parámetro importante son las dimensiones de la entrada de la red. El valor de éste se mantuvo del modelo LeNet-5 donde las imágenes tienen tamaño de 32×32 , con la adición de que las imágenes en nuestra red tienen tres canales porque se encuentran en el espacio de color RGB.

Figura 3.2: Arquitectura LeNet-5



LeNet-5 consiste de dos capas de convolución seguidas de dos capas completamente conectadas, donde la última produce la salida de la red. Por facilidad sólo se experimentó con modelos de una y dos capas de convolución. Las dimensiones de los kernels que se probaron fueron de 7×7 , 5×5 y 3×3 . En el caso de los modelos con dos capas de convolución se pusieron por pares los tamaños de los kernels; 7×7 con 5×5 , y 5×5 con 3×3 , donde los kernels de dimensión mayor iban en la primera capa de convolución. Para estructuras con una capa de convolución los kernels se mantuvieron con una dimensión de 5×5 . El número de mapas de características que se deberían producir en cada capa de convolución se tomó de entre los valores: 8, 16, 32, emparejando 8 con 16 y 16 con 32. Donde 8 y 16 son el número de mapas obtenidos de la primera capa de convolución y 16 y 32 para la segunda. Queremos mantener el tamaño de los mapas de características del mismo que el de las entradas de la convolución, esto se logra con un borde de ceros llamado *same*, el cual se asegura que las salidas y entradas tengan las mismas dimensiones. La zancada

o stride se mantiene constante con un valor de 1. Para la parte del pooling queremos ir disminuyendo las dimensiones de las entradas a la mitad después de cada convolución, por lo que se ocupa un agrupamiento máximo de 2×2 .

En cuanto a las capas completamente conectadas, se evaluaron modelos con una capa oculta y una capa de salida y estructuras con únicamente la capa de salida. El número de unidades en la capa oculta se escogió de entre los valores 1024, 512 y 128. En la capa de salida la primera unidad corresponde a la categoría de *zona de trabajo*, la segunda a *salida de emergencia*, la tercera a *cubículo* y la última a *cancha de fútbol*.

La función de activación que se utiliza en la capa de convolución es la función ReLU. En las arquitecturas con una capa oculta, la salida de ésta también se pasa por una función de activación ReLU. Para la salida de la red, queremos una distribución de probabilidad sobre un conjunto de etiquetas mutuamente excluyentes. Esto es, queremos que la salida de la unidad i en la capa de salida nos de la probabilidad p_i de que una imagen que entra a la red pertenezca a la categoría i . Para nuestro caso en específico la salida sería un vector $\mathbf{p} = (p_1, p_2, p_3, p_4)$ donde $\sum_{i=1}^4 p_i = 1$. El cálculo de los valores de \mathbf{p} se logra usando una la función *softmax*.

Remark. La función softmax se emplea para mapear un vector $\mathbf{z} \in \mathbb{R}^k$, en un vector $\sigma(\mathbf{z})$ de dimensión k de valores reales en el rango $(0, 1]$ cuya suma es igual a 1. La función está dada por:

$$\sigma : \mathbb{R}^k \rightarrow \left\{ \sigma \in \mathbb{R}^k \mid \sigma_i > 0, \sum_{i=1}^k \sigma_i = 1 \right\}$$

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{i=1}^k e^{z_i}} \text{ para } j = 1, \dots, k$$

El tipo de función de costo depende principalmente de la tarea a realizar, para nosotros una tarea de clasificación en múltiples clases. La función de costo asociada con este problema es la *entropía cruzada*.

Remark. La entropía cruzada mide la diferencia entre dos distribuciones de probabilidad, y está definida como sigue:

$$L(\mathbf{y}, \mathbf{p}) = - \sum_n y_n \log(p_n) \quad n \in [1, N]$$

donde \mathbf{y} denota la salida deseada y \mathbf{p} contiene las probabilidades para categoría. Si hay N unidades en la capa de salida, entonces $\mathbf{p}, \mathbf{y} \in \mathbb{R}^N$. \mathbf{y} y \mathbf{p} deben ser dos

Tabla 3.1: Resumen de los parámetros del diseño de la CNN con los que se experimentó.

Parámetro de diseño	Valores probados
Capas	1 convolución 1 completa
	1 convolución 2 complejas
	2 convolución 1 completa
	2 convolución 2 completas
Capas de convolución	
Tamaño del kernel	$7 \times 7, 5 \times 5, 3 \times 3$
Borde de ceros	Same
Zancada	1
Mapas de características	8, 16, 32
Pooling	Agrupamiento máximo (vecindades de 2×2).
Función de activación	ReLU
Capas completamente conectadas	
Salidas	1024, 512, 128, 4
Funciones de activación	ReLU y Softmax
Aprendizaje	
Función de costo	Entropía cruzada
Optimizador	Descenso de gradiente
Modo de actualización de pesos	Por lotes
Tamaño del lote	100, 200, 300
Épocas	500, 1000, 2000
Tasa de aprendizaje	0.0001, 0.001, 0.01

distribuciones de probabilidad.

Por último se utiliza el algoritmo de retropropagación para minimizar el error actualizando los pesos de los kernels y de las capas completamente conectadas. El modo de actualización es por lotes, donde el tamaño de los lotes fue de elegido entre los valores 100, 200 y 300; y para el número de épocas entre 500, 1000 y 2000. El último hiperparámetro para el algoritmo de aprendizaje es la tasa de aprendizaje del que se experimentó con las constantes 0,001, 0,001 y 0,01.

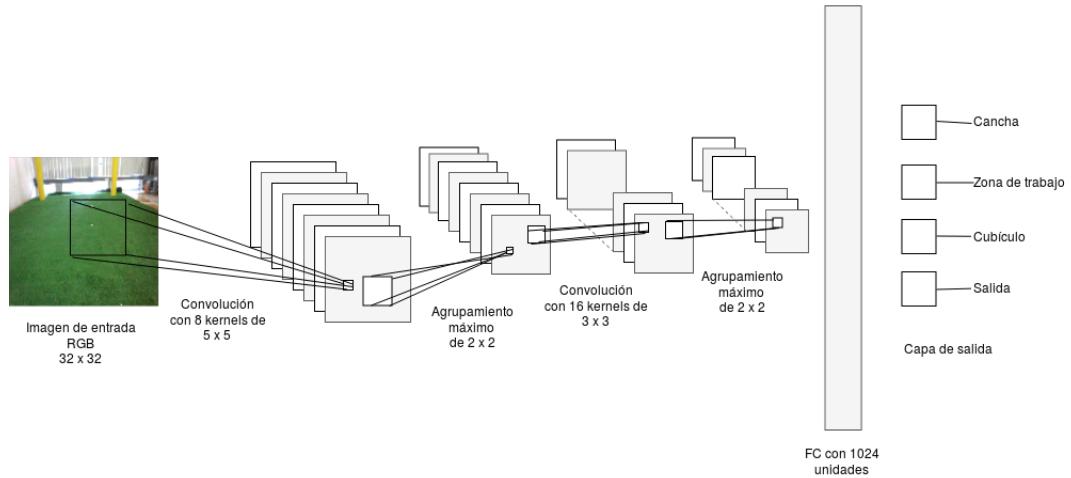
A partir de estos parámetros se pueden formar 756 arquitecturas distintas. La estructura de la red tiene cuatro formas posibles, dos capas de convolución y dos completamente conectadas, dos de convolución y una completa, una de convolución y dos completas y por último una de convolución y una completa.

Para estructuras con dos capas de convolución y dos capas completamente conectadas tenemos dos pares de dimensiones para el kernel y dos pares para el número de

mapas de características. Podemos escoger 3 valores para el número de unidades en la capa oculta. Y tenemos 3 valores distintos para la tasa de aprendizaje, el número de épocas y el tamaño del lote. Por lo que tenemos un total de $2 \times 2 \times 3 \times 3 \times 3 = 324$ modelos posibles.

Para los siguientes casos: dos capas de convolución y una capa completamente conectada, una capa de convolución y dos capas completamente conectadas y una capa de convolución y una capa completamente conectada, haciendo las mismas operaciones tenemos 108, 243 y 81 respectivamente. Un ejemplo de una de las posibles arquitecturas se muestra en la figura 3.3.

Figura 3.3: Ejemplo de una CNN con dos capas de convolucionales y dos completamente conectadas. El tamaño de los kernels en la primera capa de convolución es de 5×5 y de la segunda de 3×3 . Los mapas de características que se obtienen de la primera y segunda capa de convolución son 8 y 16, respectivamente. Se utiliza un agrupamiento máximo para reducir el tamaño de la imagen de entrada hasta un 75 por ciento. Finalmente la capa oculta conectada con la capa de salida cuenta con 1024 unidades.



3.2. Implementación del servicio sobre CloudNAO

Para crear y probar cada una de las arquitecturas que se describieron en la sección anterior se utilizó la biblioteca de TensorFlow. Se construyó una clase que pudiera crear gráficas de modelos con los cuatro casos antes descritos y luego se automatizó el proceso de entrenamiento de todas las posibles arquitecturas, variando los parámetros. Así, cada instancia de la clase, es un modelo diferente. En esta sección se

describe la implementación de las CNN, su entrenamiento, evaluación y lanzamiento sobre la API REST de CloudNAO.

El programa en TensorFlow encargado del entrenamiento de la CNN se puede dividir en tres partes principales; la primera es la carga las imágenes de cada conjunto a partir de su ruta y su etiquetado, en la segunda se crean las gráficas de cómputo y por último se ejecutan esas gráficas de cómputo.

Para agregar el modelo a la API de CloudNAO sólo se añadió un nuevo módulo al recurso `/vision`, en donde se carga la gráfica de cómputo con los mejores resultados.

3.2.1. Hardware y software

El hardware sobre el que se realizó el entrenamiento de las diferentes arquitecturas fue un equipo con las siguientes características:

- Procesador: Intel Core i5-7300HQ CPU 2.50GHz x 4.
- Memoria RAM: 8GB.
- GPU: NVIDIA GeForce GTX 1050 2GB.
- SSD: 250 GB.

Como ya se explicó, la biblioteca para desarrollar el modelo en Python fue TensorFlow, la cual cuenta con soporte para la plataforma de cómputo paralelo CUDA (Arquitectura Unificada de Dispositivos de Cómputo), desarrollada por NVIDIA. La versión de CUDA y de TensorFlow (con soporte para GPU) fueron 8.0 y 1.4.0, respectivamente.

Además de TensorFlow, las otras dos bibliotecas importantes son `numpy` y `opencv`, principalmente para la carga de las imágenes en un formato que pudiera recibir la CNN.

3.2.2. Preparación del conjunto de datos

Contamos con un conjunto de entrenamiento de 6000 imágenes y un conjunto de prueba de 2112. Como estamos utilizando un algoritmo de aprendizaje supervisado, se necesitan etiquetas u objetivos para cada imagen. La representación de esas etiquetas se hizo con vectores \mathbf{y} en codificación one-hot, con $y_n = 1$ si la imagen pertenece a la clase n y $y_i = 0$ para todos los demás casos. Los vectores posibles de acuerdo las cuatro categorías son los siguientes:

- $(1, 0, 0, 0)$ si la imagen es de la categoría *zona de trabajo*.

- (0, 1, 0, 0) si la imagen es de la categoría *salida*.
- (0, 0, 1, 0) si la imagen es de la categoría *cubículo*.
- (0, 0, 0, 1) si la imagen es de la categoría *cancha*.

Esta codificación se puede interpretar como una distribución de probabilidad sobre las cuatro clases.

Se definió un módulo `cnn_inoor_classifier_model` con una clase `CNNClassifierLAR` que encapsula los métodos para la carga de las imágenes, su procesamiento, la definición de la gráfica de cómputo y la ejecución. Dentro de la clase el método `get_training_and_test_images()` carga y procesa las imágenes para definir los conjuntos de entrenamiento y de prueba para nuestro modelo. Primero se obtiene la ruta de cada imagen y se crea la lista con las etiquetas (vectores one-hot) de cada imagen dependiendo de la carpeta donde se encontraba, luego con `opencv` se cargan las imágenes a la memoria como arreglos de `numpy`, se redimensionan a un tamaño de $32 * 32$ y se normalizan los valores de los pixeles. Al final se tienen dos arreglos de `numpy` donde cada elemento es un arreglo de tres dimensiones que representa una imagen. El código de las operaciones dentro del método son las siguientes:

```
# Carga del conjunto de entrenamiento
training_paths = np.array(list_files_in_directory('dataset/training_set/*'))
np.random.shuffle(training_paths)
labels_training_set = get_labels_from_path(training_paths)
training_images = np.array([cv2.cvtColor(cv2.resize(cv2.imread(file_name),
    (32, 32)), cv2.COLOR_BGR2RGB) / 255 for file_name in training_paths])
# Conjunto de prueba
testing_paths = np.array(list_files_in_directory('dataset/test_set/*'))
np.random.shuffle(testing_paths)
labels_test_set = get_labels_from_path(testing_paths)
testing_images = np.array([cv2.cvtColor(cv2.resize(cv2.imread(file_name),
    (32, 32)), cv2.COLOR_BGR2RGB) / 255 for file_name in testing_paths])
```

3.2.3. Construcción del modelo en TensorFlow

Definición del modelo

Para la implementación de la CNN en TensorFlow se utilizó una clase que representara una arquitectura. Esto fue para que experimentar con una estructura y parámetros diferentes sólo implicara instanciar un objeto de la clase. La clase se llama `CNNClassifierLAR` y está dentro del módulo `cnn_inoor_classifier_model`.

El módulo anterior además de la clase contiene múltiples funciones auxiliares para facilitar la definición de los parámetros de aprendizaje, la operaciones entre los tensores, la obtención de las rutas de las imágenes y la generación de los vectores one-hot de sus etiquetas.

Se pueden construir hasta cuatro tipos distintos de arquitecturas, con dos capas convolucionales con una o dos completamente conectadas, y con una capa de convolución y uno o dos completamente conectadas. La construcción de éstas se hace con los métodos `create_graph_2_convo_layers()` y `create_graph_1_convo_layer()` que reciben como argumento una bandera para construir su gráfica de cómputo con dos o una capa completamente conectada. A continuación se desglosan los pasos del método para crear la gráfica con dos capas de convolución.

1. La creación de la primera de convolución, recibe como entradas el placeholder con las imágenes y a la función se le pasan como parámetros una lista con las dimensiones de los kernels que se aplican y el número de mapas de características. Por ejemplo un lista [5, 5, 3, 16] indica que se deben aplicar kernels de 5*5 a una entrada con 3 canales (la imagen RGB), para obtener 16 mapas de características, después se aplica un agrupamiento para disminuir las dimensiones de los 16 mapas a la mitad:

```
convo_1 = convolutional_layer(input_x_ph, shape)
convo_1_pooling = max_pool_2by2(convo_1)
```

2. Luego sigue la otra capa de convolución que tiene como entrada las salidas de la capa anterior. Se aplica de nuevo un agrupamiento para reducir las al 50 % las dimensiones de la imagen:

```
convo_2 = convolutional_layer(convo_1_pooling, shape)
convo_2_pooling = max_pool_2by2(convo_2)
```

3. Se redimensionan los mapas de características para tener una red neuronal donde cada unidad es un elemento dentro del mapa de características. Esto es equivalente a aplicar una convolución con kernel de 1×1 . Se conectan las unidades de la capa a una nueva capa oculta completamente conectada.

```
convo_2_flat = tf.reshape(convo_2_pooling, [-1, 8 * 8 * last_maps_of_
    ↵features])
full_layer_one = tf.nn.relu(normal_full_layer(convo_2_flat, units_
    ↵fc))
```

4. La última capa es el vector predicho, que se pasa por la función softmax para medir el error con la entropía cruzada. Finalmente se tiene la operación `train` que minimiza la función de costo utilizando el algoritmo de retropropagación:

```

y_pred = tf.identity(normal_full_layer(full_layer_one, 4))
cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_
    ↪logits(labels=y_true_ph, logits=y_pred))
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_
    ↪rate)
train = optimizer.minimize(loss=cross_entropy, global_step=tf.train.
    ↪get_global_step())

```

Ejecución

La ejecución de la gráfica de cómputo de una instancia es con el método `run_graph()`. Se pasan como parámetro el número de épocas y el tamaño del lote para el aprendizaje. Este método ejecuta la operación `train` que a su vez corre todas las operaciones que la preceden. Cada 100 épocas se evalúa la precisión del modelo y se imprimen: el número de época, el tiempo transcurrido desde que se inició el entrenamiento, el error de entrenamiento, el error de prueba y la precisión.

```

class cnn_indoor_classifier_model.CNNClassifierLAR(shape_convolutional_layers,
units_fc, learning_rate)

```

La clase que representa una arquitectura de red convolucional con a lo más dos capas de convolución y dos capas completamente conectadas.

La red está creada para específicamente aceptar entradas de dimensiones de 32 pixeles por 32 pixeles por 3 canales. El constructor recibe tres parámetros una lista con las dimensiones de los kernels, entradas de la convolución y número de mapas de características, un entero con el número de unidades en la capa oculta y un valor flotante que es la tasa de aprendizaje. En el siguiente ejemplo se crea una red con dos capas de convolución, una capa oculta con 2048 unidades y una tasa de aprendizaje de 0.01. La ejecución de la gráfica para el aprendizaje por lotes recibe como parámetros 200 y 3000, el tamaño del lote y el número de épocas, respectivamente.

```

>>> from cnn_indoor_classifier_model import CNNClassifierLAR
>>> clasificador = CNNClassifierLAR([[5, 5, 3, 16], [3, 3, 16, 32]], 2048,
    ↪0.01)
>>> clasificador.create_graph_2_convolutional_layers()
>>> clasificador.run_graph(200, 3000)
i; time; training loss; test loss; accuracy
0; 0.7688136100769043; 1.5798137187957764; 1.4590167999267578; 0.
    ↪13778409361839294
...

```

```

3000;68.31876397132874;0.0036288381088525057;0.11022621393203735;0.
˓→9692234992980957
Time 68.3188087940216

```

3.2.4. Resultados

Los modelos con la precisión más alta de cada una de las cuatro estructuras mencionadas previamente se resumen en la tabla 3.2. Las abreviaciones Convo y FC denotan a capas convolucionales y completamente conectadas, respectivamente. γ denota la tasa de aprendizaje, y el tiempo es la duración del entrenamiento en segundos.

Tabla 3.2: Los modelos con la precisiones más altas en cada tipo de estructura.

Modelo	Épocas	γ	Lote	Kernels	Mapas de características	Unidades de la capas FC	Precisión	Tiempo
(1) 2 Convo 2 FC	2000	0.01	200	5×5 , 3×3	16, 32	1024, 4	0.97917	168.29
(2) 2 Convo 2 FC	2000	0.01	300	5×5 , 3×3	16, 32	1024, 4	0.97917	191.71
(3) 2 Convo 1 FC	2000	0.01	300	7×7 , 5×5	16, 32	4	0.97159	79.73
(4) 1 Convo 2 FC	2000	0.01	200	5×5	16	512, 4	0.97775	119.13
(5) 1 Convo 1 FC	2000	0.01	100	5×5	32	4	0.96117	45.83

Los modelos con la mayor precisión sobre todos son el modelo uno y dos con una proporción de valores correctamente clasificados de 0.97917. Se puede ver que los parámetros que comparten todas las arquitecturas son el número de épocas y la tasa de aprendizaje, además de que en sus kernels aparece al menos uno con una dimensión de 5×5 .

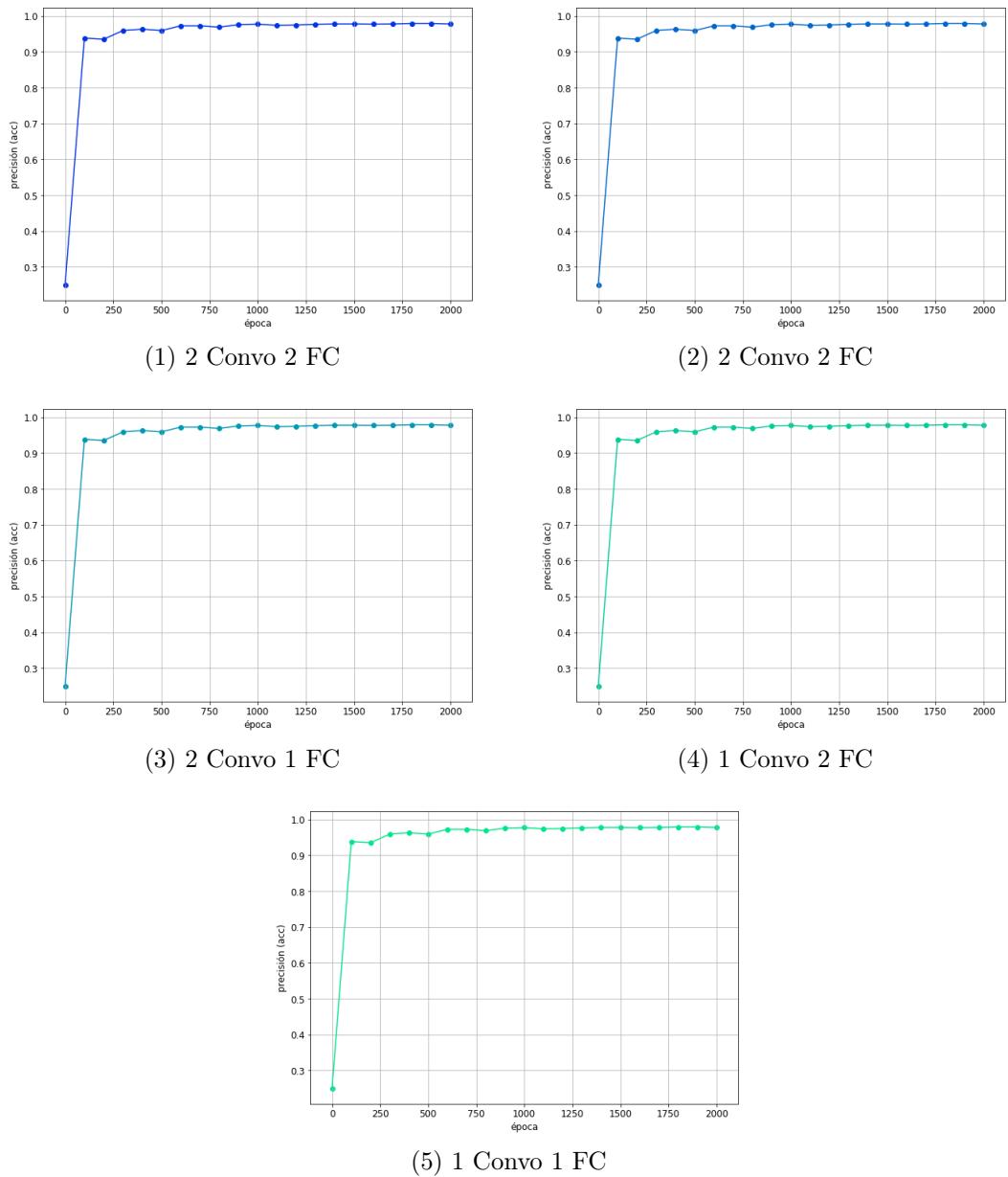


Figura 3.4: La precisión de cada modelo con respecto a las épocas.

Pruebas sobre el robot NAO

Para finalizar la implementación del modelo sobre la arquitectura se debe integrar uno de los cinco modelos descritos en la tabla 3.2 sobre la API REST de CloudNAO y luego hacer peticiones con el robot a ésta. Los modelos elaborados en TensorFlow contienen la gráfica de cómputo y los valores de los parámetros que se han en-

trenado. Estos datos están contenidos en varios archivos que se pueden guardar para entrenamientos posteriores o para realizar inferencias sobre un modelo cuyas variables han sido aprendidas.

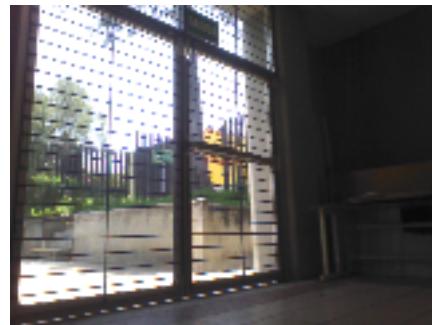
Se eligió el modelo con la precisión más alta y con el menor tiempo de entrenamiento que es el modelo (1). Dentro de la API REST, la clase que se encarga de restaurar la gráfica y valores del modelo es `ImageClassifier` dentro del módulo `indoor_scenes_classifier` del paquete `tf_models`. En el apartado *Modelos de Tensorflow (tf_models)* de la sección 2.1.2 se describe cómo utilizar esta clase.

Por otra parte el robot NAO o cualquier otro dispositivo cliente se comunica con la API a través de una biblioteca que permita hacer peticiones HTTP. En el caso del robot ocupamos la biblioteca `requests` de Python. Para obtener una imagen del robot codificada en base 64 se utilizó la clase `Robot` del módulo `nao_robot` descrito en la sección 2.1.5. Es la cadena que representa la imagen la que se envía a la API REST usando `requests`.

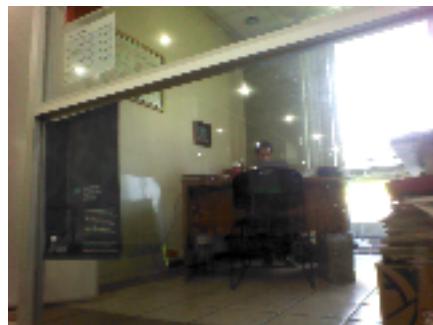
En la figura 3.5 se muestran algunas imágenes de 160×120 pixeles y las respuestas enviadas por el servicio de clasificación de escenarios dentro del recurso `vision` de la API REST de CloudNAO.



(a) Clase = zona de trabajo, predicción = zona de trabajo. Tiempo = 3.02



(b) Clase = salida, predicción = salida. Tiempo = 3.12



(c) Clase = oficina, predicción = salida. Tiempo = 2.78



(d) Clase = cancha, predicción = cancha. Tiempo = 3.57

Figura 3.5: Predicciones hechas enviando imágenes del robot a la API REST. El tiempo son los segundos que se tardó en enviar la solicitud y en recibir la respuesta.

Conclusiones

Anexos

Bibliografía

- [1] Albert Einstein. Zur Elektrodynamik bewegter Körper. (German) [On the electrodynamics of moving bodies]. *Annalen der Physik*, 322(10):891–921, 1905.
- [2] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The TeX Companion*. Addison-Wesley, Reading, Massachusetts, 1993.
- [3] Donald Knuth. Knuth: Computers and typesetting.