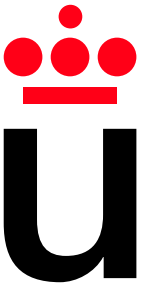


Universidad
Rey Juan Carlos

Escuela Técnica Superior
de Ingeniería Informática

Grado en Ingeniería del Software

Curso 2023-2024

Trabajo Fin de Grado

MÉTODOS DE APRENDIZAJE AUTOMÁTICO PARA LA CLASIFICACIÓN DE COMPORTAMIENTOS ESTEREOTIPADOS

Autor: Gonzalo Ortega Carpintero
Tutor URJC: Alberto Fernández Gil
Tutor CSIC: Pablo Jercog

Agradecimientos

A Paula Peixoto Moledo por guiarme en todo el proceso de lectura, desarrollo y redacción de este trabajo y a Jesús González Gironda por mostrarme la oportunidad de realizar este trabajo en el CSIC.

Resumen

A lo largo de la historia, en ciencia en general y en Neurociencia de sistemas en particular, se han realizado todo tipo de experimentos en los cuales los experimentadores tenían que llevar un control manual del proceso, realizando mediciones limitadas sujetas a la interpretación y sesgo de cada experimentador, que necesitaban ser analizadas individualmente. Con el reciente auge de los métodos de aprendizaje automático, ahora es posible recopilar y analizar muchos más datos con un mismo presupuesto, utilizando diversas técnicas para analizar y obtener resultados de los experimentos de forma automática.

Este trabajo se enfoca en distinguir de forma automática los diferentes tipos de comportamientos que realiza un animal en su caja hábitat a lo largo de una sesión de video. Además, busca diferenciar animales control de aquellos que están bajo los efectos de un bloqueador de NMDA. Para ello, primero se realiza una introducción teórica a algoritmos de aprendizaje no supervisado: PCA, agrupación por k-medias, agrupación aglomerada, y agrupación por afinidad; y a algoritmos de aprendizaje supervisado: redes neuronales secuenciales y convolucionales. Posteriormente, se aplican estos algoritmos a datos de la postura corporal de animales a lo largo de sesiones de video procesadas con DeepLabCut, una herramienta de código abierto basada en el aprendizaje profundo, diseñada para rastrear el comportamiento animal con alta precisión.

Palabras clave:

- Neurociencia de sistemas
- Aprendizaje automático
- PCA
- k-medias
- Agrupación aglomerativa
- Agrupación por afinidad
- Red neuronal secuencial
- Red neuronal convolucional
- DeepCutLab
- Python

Índice de contenidos

Índice de tablas

Índice de figuras

Índice de códigos

1 Introducción

Este documento recoge el Trabajo de Fin de Grado (TFG) de Gonzalo Ortega Carpintero, incluido en el itinerario del grado en Ingeniería del *Software* de la Universidad Rey Juan Carlos.

En este primer capítulo se presenta la motivación que ha dado lugar a la idea y desarrollo de este TFG en la Sección 1.1. En la Sección 1.2 se introduce el objetivo principal del trabajo y se enumeran los objetivos parciales que se han propuesto para conseguirlo. Por último, en la Sección 1.3 se hace un repaso a la metodología seguida para realizar el trabajo.

1.1. Motivación

La Neurociencia de sistemas es una rama de la neurociencia que estudia cómo los diferentes componentes del sistema nervioso, desde las neuronas individuales hasta las redes neuronales, interactúan y funcionan conjuntamente para producir el comportamiento y los procesos mentales [1]. Para el estudio del funcionamiento del cerebro pueden utilizarse multitud de técnicas diferentes, pero para evitar la intrusión en el sujeto de estudio de muchas de ellas, muchos experimentos e investigaciones se basan en el estudio del comportamiento. Es la etología la encargada de este tipo de estudio en animales y en la cual, históricamente, la categorización de comportamientos dependía fuertemente de la observación manual. Eso ha conllevado siempre una excesiva cantidad de tiempo y las desventajas de los posibles errores de percepción humanos, limitando la cantidad y la calidad de los análisis

a realizar.

Gracias a los recientes avances en algoritmos de aprendizaje automático, hoy en día es posible computar multitud de datos de forma simultánea, agilizando y automatizando análisis como el ilustrado previamente.

Este Trabajo de Fin de Grado se ha realizado a lo largo de una estancia de prácticas académicas en el Jercog's Team, un equipo de investigación perteneciente al Instituto Cajal, instituto de Neurociencia del Consejo Superior de Investigaciones Científicas. El equipo está centrado en el estudio de la memoria mediante experimentos con ratones, y uno de sus proyectos abiertos consistía en poder elaborar una herramienta para clasificar automáticamente los comportamientos estereotipados de los ratones. Estos son comportamientos cortos, repetitivos y con cierta tendencia a generar patrones, tales como rascarse, caminar en círculos o levantarse a dos patas, ejecutados sin ninguna finalidad, y usualmente inducidos por estar en entornos cerrados y artificiales.

Para la ejecución de uno de sus últimos experimentos utilizan ratones tratados con anticuerpos bloqueadores de NMDA (*N-Metil-D-aspartato*), restringiendo la capacidad de los ratones para almacenar memoria a largo plazo. En las sesiones de experimentación los ratones tratados ejercen las mismas tareas que ratones control, tratados con un suero placebo, y en las grabaciones hay ratones de ambos tipos. A ojo de un experimentador, es complejo determinar a ciegas si un ratón está bajo los efectos de NMDA o es control, por lo que es interesante la posibililidad de contar con una herramienta que sea capaz de distinguir unos de otros a partir de los videos de las sesiones.

Durante la estancia se han valorado multitud de técnicas de análisis y procesado de datos, haciendo hincapié en métodos de aprendizaje automático para tratar de completar el proyecto gracias a los últimos avances en computación.

1.2. Objetivos

El objetivo de este trabajo es distinguir de forma automática, mediante la clasificación de los comportamientos o de forma directa, si un animal está bajo los efectos de un bloqueador de NMDA.

1.2.1. Hipótesis

Para contrastar nuestro objetivo hemos planteado la siguiente hipótesis:

Dada la postura corporal de un ratón en su caja a lo largo de una sesión de video, mediante técnicas de aprendizaje automático, es posible diferenciar difer-

entes comportamientos del ratón, e identificar si está bajo los efectos de un bloqueador de NMDA.

1.2.2. Objetivos parciales

Con el fin de desglosar el objetivo principal, y de poder corroborar o refutar nuestra hipótesis, hemos planteado varios objetivos parciales.

- Realizar una introducción a técnicas de aprendizaje automático aplicables al procesado de datos en el análisis de experimentos con animales.
- Preparar y preprocesar datos provenientes de archivos de video para su posterior análisis.
- Aplicar algoritmos de agrupación para separar fragmentos de video y clasificarlos en función del comportamiento que esté realizando el animal.
- Aplicar algoritmos no supervisados para distinguir animales control de animales tratados.
- Aplicar algoritmos supervisados para distinguir animales control de animales tratados.

1.3. Metodología

El primer paso para la realización de este trabajo fue la lectura de literatura relacionada, tanto de artículos de Neurociencia de sistemas que abordaban problemas similares, como de manuales de diseño y programación de técnicas de aprendizaje automático. Tras evaluar que métodos y herramientas utilizar, se realizó el preprocesado de los datos de los videos de las sesiones rastreando diferentes partes de los animales con DeepLabCut y filtrando e interpolando los datos obtenidos para prepararlos para el análisis. Con todo preparado, pasamos al análisis de los datos, realizando técnicas de clasificación manual, de aprendizaje no supervisado y de aprendizaje supervisado.

En el laboratorio hemos seguido una metodología ágil, con numerosas iteraciones de lectura de literatura, desarrollo de código en Python, y pruebas de los métodos implementados. Cada cierto tiempo realizábamos reuniones de laboratorio para actualizar al resto de integrantes del laboratorio del estado del trabajo, recibiendo comentarios e ideas para seguir iterando.

El trabajo se divide en dos capítulos principales, en el Capítulo 2 abordaremos el estudio teórico de ciertos algoritmos de agrupamiento, reducción de dimensionalidad y de clasificación supervisada. Todo ello desde un punto de vista teórico con

ejemplos fáciles de visualizar para ayudar al lector a comprender en profundidad los métodos.

En el Capítulo 3, desarrollaremos paso a paso el análisis real ejercido durante la estancia en el Instituto Cajal. El principal objetivo durante estos meses ha sido buscar la mejor forma de, dados una serie de videos de ratones en su caja, detectar automáticamente diferentes tipos de comportamientos. El fin último del proyecto es estudiar la posibilidad de clasificar dos grupos de animales, control y medicados, basándose en los comportamientos estereotipados.

2

Fundamentos teóricos

En este Capítulo vamos a dar una introducción a los algoritmos de aprendizaje automático que más adelante usaremos para realizar nuestro análisis con datos reales de laboratorio.

En la Sección 2.1 vamos a tratar algunos algoritmos no supervisados, tanto para la reducción de la dimensión de los datos y su visualización, como para la agrupación de los datos en distintas clases, con un número de ellas tanto preestablecido como generado automáticamente a partir de ellos. Posteriormente, en la Sección 2.2 vamos a introducir algoritmos de clasificación supervisada centrándonos en las redes neuronales, tanto secuenciales como convolucionales.

2.1. Métodos no supervisados

Los algoritmos de aprendizaje automático no supervisados se utilizan cuando no se conoce la salida esperada. Al algoritmo de aprendizaje se le otorgan los datos de entrada y se le pide extraer información de estos datos. Las principales aplicaciones de estos algoritmos, las cuales vamos a aprovechar, son la agrupación de datos y la reducción de dimensionalidad de las variables de los mismos. Esa última es usada principalmente para poder hacer representaciones de datos multidimensionales, los cuales serian complejos de visualizar de otra forma.

La principal pega que pueden tener estos algoritmos es que, si bien no siempre son capaces de identificar conocimiento dados los datos utilizados, cuando

lo obtienen, no siempre es el conocimiento que esperábamos obtener. Póngase el ejemplo de un algoritmo que tratase de agrupar rostros de personas iguales. Al no darle a priori ningún tipo de salida de ejemplo, el algoritmo puede acabar clasificando si los rostros están de frente o de lado, no precisamente lo que esperábamos. Es por ello que estos algoritmos cuentan con diversidad de parámetros para ajustarlos a nuestras necesidades, tratando de realizar la agrupación deseada.

En esta sección vamos a estudiar a fondo tres tipos de algoritmos de agrupación: el agrupamiento por k-medias, el agrupamiento aglomerativo, y el agrupamiento por afinidad. Además, estudiaremos también el principal algoritmo de reducción de dimensiones, el análisis de componentes principales, PCA, de sus siglas en inglés. Los principales ejemplos y explicaciones de los algoritmos han sido inspirados por los dados en [2].

Código 2.1: Generar datos artificiales de prueba.

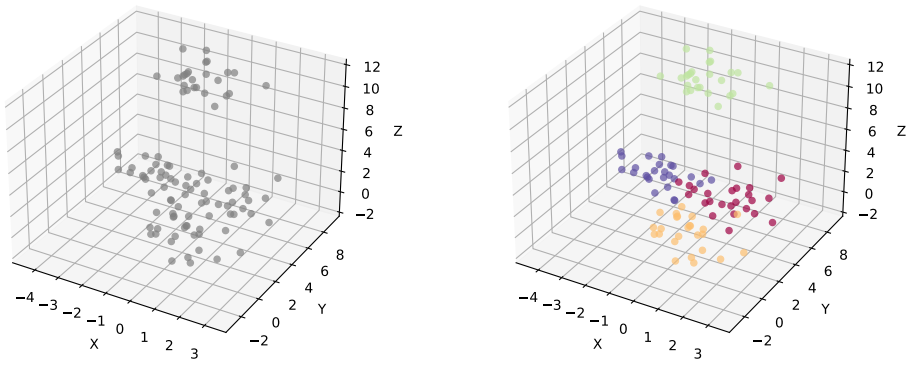
```
1  from sklearn.datasets import make_blobs
2
3  X, y = make_blobs(n_samples=100, centers=4,
4  n_features=3, random_state=0)
```

Hemos creado un conjunto de datos sintéticos a los cuales poder aplicar todos los algoritmos que vamos a explicar para poder visualizar gráficamente sus resultados. Con la función `make_blobs` de Scikit-learn hemos aleatorizado 100 puntos tridimensionales agrupados en cuatro clases fácilmente diferenciables. Scikit-learn es una de las bibliotecas de Python especializada en aprendizaje automático que hemos utilizado. La figura 2.1 muestra una representación gráfica del conjunto, y el Código 2.1 muestra los parámetros que hemos utilizado para crearlo. En la tabla 3.1 está el enlace para poder ejecutar el código en el *backend* de Google.

2.1.1. Análisis de componentes principales

Nuestro conjunto de datos artificiales tiene únicamente 3 dimensiones, lo que hace que sea sencillo poder representarlo gráficamente y que sea fácilmente interpretable. Sin embargo, en la vida real, así como en los datos que en el Capítulo 3, los datos tienden a tener muchas más dimensiones. Esto hace que sea un problema poder representarlos en espacios bidimensionales como este documento, y puede entorpecer también su interpretación o lectura desde otros algoritmos, ya que es probable que haya dimensiones que aporten más ruido que información a la muestra.

Una de las técnicas más utilizadas para tratar este problema es realizar un análisis de componentes principales (PCA) sobre los datos [3]. Un PCA es una transformación lineal de los datos a un nuevo sistema de coordenadas para facil-



(a) (b)

Figura 2.1: Datos artificiales generados para probar algoritmos de agrupamiento. Consisten en 100 puntos tridimensionales agrupados equilibradamente al rededor de 4 centros aleatorios. 2.1a En gris los puntos generados sin asignar ninguna etiqueta. Estos serán los datos que recibirán los algoritmos para procesar y agrupar. 2.1b Datos etiquetados según el grupo al que pertenecen. Esto nos ayudará a verificar la eficacia de nuestros algoritmos.

itar la identificación de las direcciones que contienen más variabilidad. A estas direcciones, que consisten en una combinación lineal de las variables originales, se les denomina *componentes principales*.

El primer paso para realizar un PCA es normalizar los datos para que cada una de las variables originales tenga media 0 y varianza 1. Para ello, denotando $\mathbf{X} = (X_1, ..., X_n)$ a nuestros datos a analizar, donde cada X_i con $i \in 1, ..., n$ es una de las variables de nuestros datos, y μ_{X_i} y σ_{X_i} son la media y desviación típica de cada una de las variables, computamos

$$Z_i = \frac{X_i - \mu_{X_i}}{\sigma_{X_i}}, \tag{2.1}$$

para cada i, obteniendo $\mathbf{Z} = (Z_1, ..., Z_n)$ el conjunto de todas nuestras variables normalizadas. Con esto se estandariza la escala de las variables de nuestros datos, evitando que las variables con escalas mayores dominen al resto. En el Código 2.2 realizamos la normalización de nuestros datos artificiales utilizando la clase `StandardScaler` de Scikit-learn.

Una vez todos nuestros datos están en la misma escala estamos en disposición de realizar el PCA propiamente dicho. Para ello calculamos la matriz de covarianza Σ de nuestros datos normalizados. Σ es una matriz de dimensión $n \times n$

Código 2.2: Escalar y estandarizar los datos.

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
scaler.fit(X)
Z = scaler.transform(X)
```

con

$$\Sigma_{ij} = \text{Cov}(Z_i, Z_j) = E[Z_i Z_j] - E[Z_i]E[Z_j], \tag{2.2}$$

para $i, j \in 1, ..., n$ siendo E el operador media. Acto seguido, calculamos los n pares de autovalores y autovectores de la matriz Σ resolviendo la ecuación

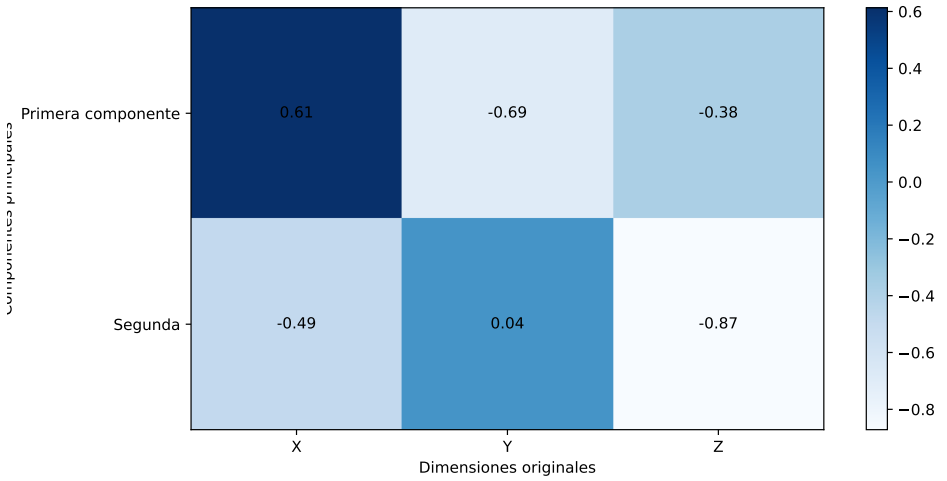
$$\Sigma \mathbf{v} = \lambda \mathbf{v}. \tag{2.3}$$

Los autovectores $\mathbf{v}$ se denominan componentes principales, y ordenándolos en orden decreciente según el valor de su autovalor λ asociado, obtenemos la lista de componentes principales deseada. Finalmente basta con quedarse con el numero deseado de componentes de la parte superior de la lista para reducir la dimensionalidad de nuestros datos maximizando la covarianza de los mismos. En el Código 2.3 hemos realizado un PCA de nuestros datos artificiales conservando las 3 componentes principales. Hemos utilizado la clase `PCA` de Scikit-learn, que encapsula todo el procedimiento que hemos explicado.

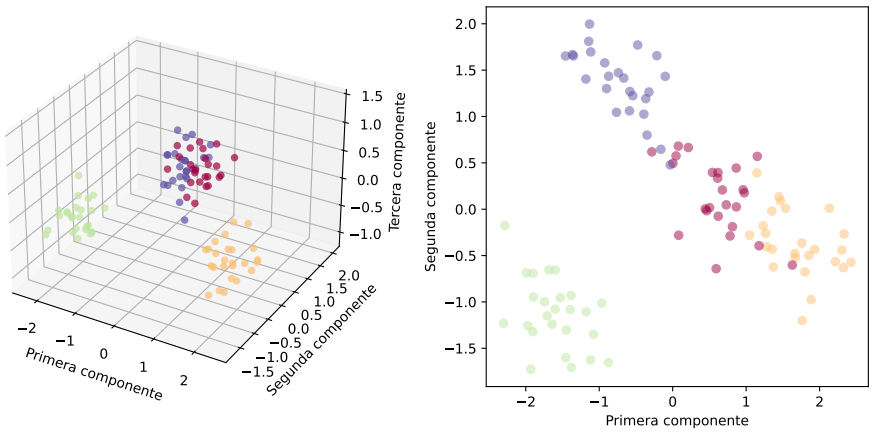
Código 2.3: Realizar un PCA.

```
from sklearn.decomposition import PCA

pca = PCA(n_components=3)
pca.fit(Z)
Z_pc = pca.transform(X)
```

En la Figura 2.2 se muestran gráficamente los puntos artificiales tras el anális de componente principales.



(a)



(b) (c)

Figura 2.2: Prueba del algoritmo de PCA. 2.2a Mapa de calor con las ponderaciones asignadas por el PCA a cada uno de las dimensiones originales. 2.2b Visualización de los datos previamente generados tras escalarlos y aplicarles la PCA. Los colores de los grupos se mantienen, pero los ejes ahora representan las componentes principales, en vez de las dimensiones originales. 2.2c Visualización de las dos primeras componentes principales sobre el plano. Este será el formato en el que visualizaremos más adelante las agrupaciones ejercidas sobre datos reales de dimensionalidad alta.

2.1.2. Agrupamiento por k-medias

El algoritmo de k-medias clasifica los datos separando los datos en k grupos de manera que la suma de las distancias al cuadrado entre los datos y el centroide del grupo al que pertenecen sea mínima.

El algoritmo comienza inicializando aleatoriamente k centroides, siendo k el número de agrupaciones que se le han dicho que realice. Estos centroides serán los centros de las agrupaciones que va a realizar, no tienen por qué pertenecer a los datos, pero sí que están contenidos en su misma dimensión. En cada iteración, el algoritmo asigna a cada punto de los datos el centroide más próximo y luego asigna a cada agrupación un nuevo centroide calculado como la media de los datos que se han asignado a dicha agrupación.

El algoritmo divide un conjunto de n elementos x_j en k agrupaciones disjuntas C_i, cada una descrita por la media μ_i de los puntos en la agrupación. Para ello se computan los centroides que

$$\underset{C}{\operatorname{argmin}} \sum_{i=1}^{k} \sum_{x_j \in C_i} (||x_j - \mu_i||^2). \tag{2.4}$$

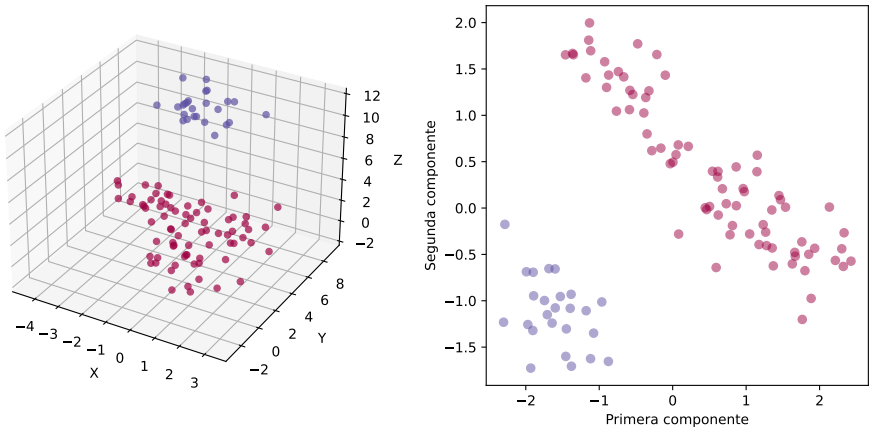
Cuando una iteración no realiza ninguna modificación de las agrupaciones generadas por los centroides de la iteración anterior, el algoritmo finaliza devolviendo las agrupaciones creadas asignando una clase a cada uno de los puntos procesados.

Para computar el algoritmo sobre nuestros datos artificiales hemos utilizado el Código 2.4 usando la clase `KMeans` de Scikit-learn. Hemos iniciado al algoritmo que separe los datos en dos clases, iniciando los centroides de forma automática. Con esto queremos ver como actúa un algoritmo dividiendo un conjunto de datos generado con 4 clases en mente, cuando se le fuerza a separar únicamente dos grupos.

Código 2.4: k-medias.

```
1  from sklearn.cluster import KMeans
2
3  kmeans = KMeans(n_clusters=2, n_init="auto")
4  kmeans_assigment = kmeans.fit_predict(X)
```

La Figura 2.3 muestra las dos agrupaciones que se han decidido crear sobre los datos artificiales, simulando lo que más adelante haremos con los datos reales.



(a) (b)

Figura 2.3: Prueba del algoritmo de k-medias. 2.3a Visualización sobre los datos sin modificar de los grupos que ha realizado el algoritmo de k-medias. 2.3b Visualización de los grupos realizados sobre los datos procesados por el PCA.

2.1.3. Agrupamiento aglomerativo

Al igual que el algoritmo de k-medias, el algoritmo de agrupamiento aglomerativo divide los datos dados en el número de grupos indicado. Para ello empieza asignando a cada punto un grupo diferente, para luego ir unificando los grupos en cada iteración según el criterio elegido, hasta alcanzar el número de grupos deseado. Este tipo de algoritmo pertenece a un tipo más amplio de algoritmos denominado agrupación jerárquica, donde además de los agrupamientos aglomerativos están las agrupaciones divisivas. Estas funcionan asignando a todos los puntos el mismo grupo y dividiendo este grupo en cada iteración hasta alcanzar el número deseado.

Los dos tipos de agrupamiento jerárquico funcionan de forma similar, y ambos varían dependiendo de la función de distancia que se utilice, tanto para dividir como para agrupar. Vamos a profundizar en el Método de Ward, que junta los grupos que minimicen la varianza mínima dentro del nuevo grupo. Así, la varianza de un grupo C de puntos $\{x_1, ..., x_n\}$ se define como

$$V(C) = \sum_{i=1}^{n} ||x_i - \mu_C||^2, \tag{2.5}$$

siendo μ_C la media de los puntos del grupo C.

Por tanto, en cada iteración el algoritmo calcula el incremento de varianza

que supondría juntar cada par de grupos A y B calculando

$$\delta(A, B) = \frac{|A||B|}{|A| + |B|} ||\mu_A - \mu_B||^2, \tag{2.6}$$

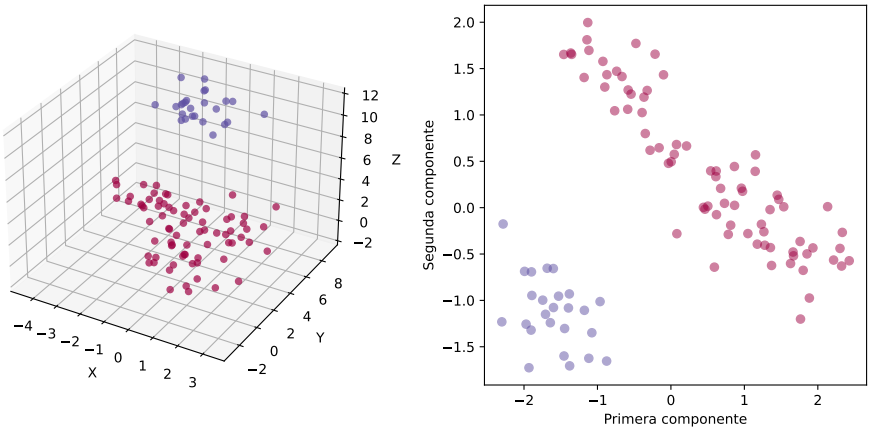
donde $|\cdot|$ denota el cardinal de los grupos y $||\cdot||$ la operación módulo.

Tras ello, unifica los dos grupos que supongan un menor incremento en la varianza y repite el proceso hasta obtener el número deseado de grupos.

Código 2.5: Agrupamiento aglomerativo.

```
1  from sklearn.cluster import AgglomerativeClustering
2
3  agg = AgglomerativeClustering()
4  agg_assigment = agg.fit_predict(X)
```

En el Código 2.5 hemos realizado un agrupamiento aglomerativo sobre nuestros puntos artificiales utilizando la clase `AgglomerativeClustering` de Scikit-learn, que por defecto, si no se le indica lo contrario, utiliza el Método de Ward en la ejecución del algoritmo. En la Figura 2.4 se muestran los resultados sobre nuestros datos de forma gráfica.



(a) (b)

Figura 2.4: Prueba del algoritmo de agrupamiento aglomerativo. 2.4a Visualización sobre los datos sin modificar de los grupos que ha realizado el algoritmo. 2.4b Visualización de los grupos realizados sobre los datos procesados por el PCA.

2.1.4. Agrupamiento por propagación de afinidad

El algoritmo de propagación de afinidad [4] crea agrupaciones mandando mensajes entre pares de puntos hasta que converge. La principal cualidad de este algoritmo es que, a diferencia de la mayoría de algoritmos de agrupamiento, no necesita saber el número de agrupaciones a realizar de antemano, sino que las genera dinámicamente. Consigue esto ya que inicializa tantos centroides como grupos se quieran obtener, el algoritmo de propagación de afinidad empieza considerando como posibles centroides a todos los puntos de la muestra.

Para ello, el algoritmo se inicializa con una matriz de afinidad S, formada por los valores de afinidad $s(i,k)$ que indican lo apropiado que sería escoger el dato k como ejemplar del dato i. Esta matriz puede inicializarse de multitud de formas, pero por defecto se utiliza el negativo de la distancia Euclídea al cuadrado. Con ella, para los puntos x_i, x_k, se tendría

$$s(i,k) = -||x_i - x_k||^2. \tag{2.7}$$

El algoritmo se ejecuta actualizando dos matrices por medio de *mensajes* hasta que se estabiliza o llega a un número máximo de iteraciones. La *matriz de responsabilidad* R con valores $r(i,k)$ cuantifica la evidencia acumulada de como de apropiado es escoger x_k para ser el ejemplar de x_i. Y la *matriz de disponibilidad* A con valores $a(i,k)$ que refleja la evidencia acumulada de como de apropiado sería que un punto x_i escogiera a un punto x_k como ejemplar.

Ambas matrices se inicializan a cero y en cada iteración se actualiza primero la matriz de responsabilidad

$$r(i,k) \leftarrow s(i,k) - \max_{k' \neq k}\{a(i,k') + s(i,k')\}, \tag{2.8}$$

después la matriz de disponibilidad para $i \neq k$

$$a(i,k) \leftarrow \min\left\{0, r(k,k) + \sum_{i' \notin \{i,k\}} \max\{0, r(i',k)\}\right\}, \tag{2.9}$$

y para el resto de casos

$$a(k,k) \leftarrow \sum_{i' \neq k} \max\{0, r(i',k)\}. \tag{2.10}$$

Tras alcanzar el criterio de parada se eligen los puntos x_k para los cuales

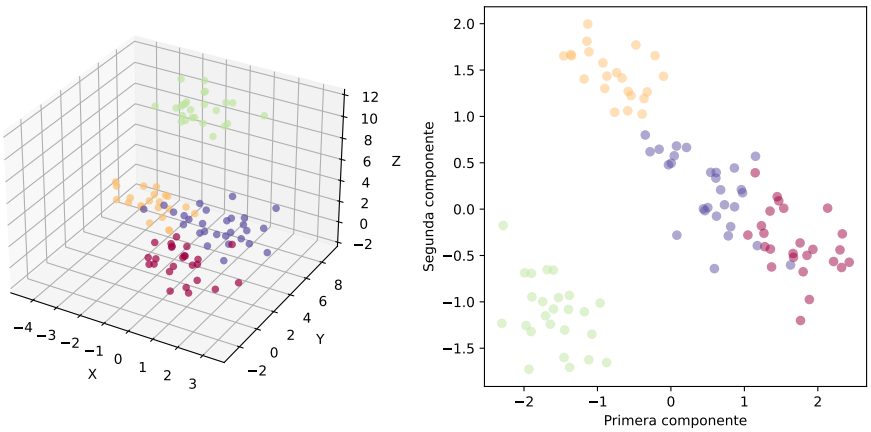
$$a(k,k) + r(k,k) > 0, \tag{2.11}$$

como puntos ejemplares determinando el número de grupos obtenidos y asignando al resto de puntos el grupo de cuyo ejemplar sean más similares.

Código 2.6: Propagación de afinidad.

```
from sklearn.cluster import AffinityPropagation

aff = AffinityPropagation()
aff_assigment = aff.fit_predict(X)
```

En el Código 2.6 hemos utilizado la clase `AffinityPropagation` de Scikit-learn para realizar un agrupamiento por propagación de afinidad sobre nuestros datos artificiales. En la Figura 2.5 vemos los resultados del agrupamiento, observando como sin haber indicado el número de grupos deseado, el algoritmo consigue diferenciar los cuatro grupos planteados por el generador de datos.



(a) (b)

Figura 2.5: Prueba del algoritmo de propagación de afinidad. 2.5a Visualización sobre los datos sin modificar de los grupos que ha realizado el algoritmo. 2.5b Visualización de los grupos realizados sobre los datos procesados por el PCA.

2.2. Métodos supervisados

Al contrario que los métodos de aprendizaje automático no supervisados, donde no sabemos de antemano la salida deseada, los métodos de aprendizaje automático supervisados permiten calibrar los modelos utilizando los datos de salida que queremos mediante el uso de datos de entrenamiento de los cuales conocemos su salida esperada. Para métodos de agrupamiento y separación de clases como los que nos interesan en este trabajo, basta con contar con un conjunto de datos previamente etiquetado con su clase real como datos de entrenamiento, permitiendo a los modelos aprender sobre la clasificación dada y posteriormente replicarla para datos nuevos.

En el laboratorio contamos con datos de sesiones etiquetadas con el estado del tratamiento de los animales, por lo que pueden servir para entrenar un método supervisado. En esta sección, contamos con los datos artificiales que hemos generado, en los cuales cada punto tiene asignado uno de cuatro grupos diferentes.

Existen multitud de métodos no supervisados, sin embargo vamos a centrarnos únicamente en el que más está siendo utilizado a día de hoy, dando buenos resultados en múltiples disciplinas para todo tipo de problemas: las redes neuronales. En esta sección vamos a introducir primero el ejemplo fundamental de redes neuronales, las redes neuronales secuenciales, para luego hablar de las redes neuronales convolucionales, que serán las que finalmente apliquemos a nuestros datos reales. Los ejemplos y explicaciones se basan en los de S. J. Prince [5] y E. Stevens et. al [6].

2.2.1. Red neuronal secuencial

Antes de profundizar en el concepto de red, es necesario definir las piezas por las cuales están formadas, las *neuronas artificiales*. Una neurona es una función que recibe un vector de entradas $\mathbf{x} = (x_1, ..., x_n)$ y asigna un peso a cada una de ellas, representados mediante un vector de pesos $\mathbf{w} = (w_1, ..., w_n)$, mediante una combinación lineal, sumando además un sesgo b,

$$z = \sum_{i=1}^{n} w_i x_i + b = \mathbf{w} \cdot \mathbf{x} + b. \tag{2.12}$$

El resultado de esta combinación lineal pasa después por una función de activación $\varphi(z)$ que lo filtra antes de obtener la salida final de la neurona

$$y = \varphi(\mathbf{w} \cdot \mathbf{x} + b). \tag{2.13}$$

La función de activación φ varía en función del problema que se quiera tratar

y del diseño de la red. Algunos de los ejemplos más comunes son la tangente hiperbólica (tanh)

$$\varphi(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}, \tag{2.14}$$

la cual vamos a usar en las pruebas de esta sección, o el rectificador linear unitario (ReLu),

$$\varphi(z) = \max(0, z), \tag{2.15}$$

el cual es ampliamente utilizado por sus buenos resultados en problemas generales, y hemos utilizado en la Sección 3.3 con nuestros datos reales.

Una vez definido el concepto de neurona podemos definir una red *neuronal secuencial completa* como un conjunto de neuronas divididas en diferentes capas. En la primera capa, la *capa de entrada*, cada neurona recibe como entrada todos los datos a procesar y manda su salida a la entrada de cada una de las neuronas de la siguiente capa. En las capas intermedias, *capas ocultas*, las neuronas reciben un vector de entrada formado por las salidas de la capa anterior, y vuelven a pasar su salida a las capas posteriores. Finalmente, la última capa, la *capa de salida*, recibe su entrada de la capa anterior y devuelve la salida final de la red. En función del número de neuronas de esta capa, la red podrá devolver más o menos información. En la Figura 2.6 se muestra un ejemplo de una posible red neuronal secuencial.

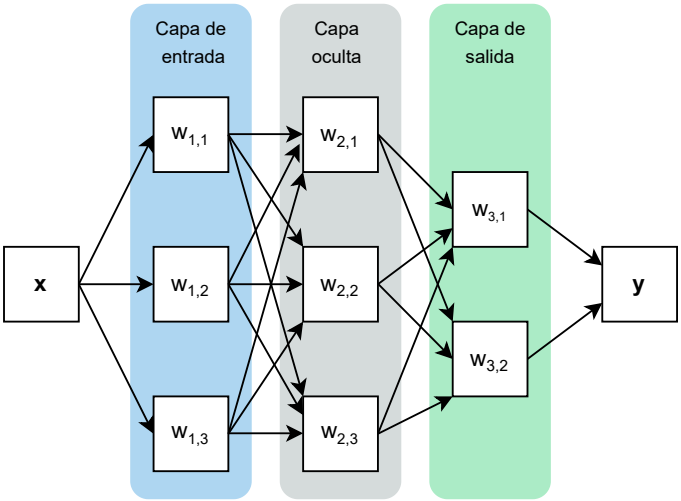


Figura 2.6: Diagrama simplificado de una red secuencial. En esta red, la capa de entrada (azul) tiene tres neuronas que reciben los datos de $\mathbf{x}$ y tras aplicar los pesos, el sesgo y la función de activación, pasan a una capa oculta (gris). Esta capa recibe todas las salidas de la capa anterior y tras repetir el procedimiento anterior, pasa su salida a la capa de salida (verde). En este caso, tiene solo dos neuronas, definiendo el tamaño de los datos de la salida $\mathbf{y}$ de la red.

Para clasificar datos mediante una red neuronal es necesario contar con un

conjunto de datos etiquetado dividido en un subconjunto de entrenamiento y otro subconjunto de validación, para poder así *entrenar* nuestra red y validar la calidad de su aprendizaje. Así, se empieza inicializando los valores de los pesos de la red a valores predeterminados, y luego se empiezan a propagar hacia delante los datos de entrenamiento calculando el error L cometido al clasificarlos comparando la salida de la red con el valor de su clasificación real

$$L = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2, \tag{2.16}$$

donde m es el número de neuronas de la capa de salida y $\hat{y}_i$ es la salida esperada en cada una de esas neuronas para cada vector de entrada propagado hacia delante.

El siguiente paso es propagar el error hacia detrás calculando el error relativo cometido por cada una de las neuronas, para poder así modificar sus pesos y sesgos. Se calcula el gradiente de L calculando la derivada parcial de L respecto a cada uno de los pesos $\mathbf{w}$ y el sesgo b y se actualizan los parámetros introduciendo un *ratio de aprendizaje* α. Denotando $\phi \in \{\text{pesos y sesgos}\}$ a cualquiera de los parámetros de las distintas neuronas, tenemos que en cada iteración cada parámetro ϕ se actualiza

$$\phi \leftarrow \phi - \alpha \cdot \frac{\partial L}{\partial \phi}. \tag{2.17}$$

Este proceso se repite hasta obtener un error aceptable y dar por concluido el entrenamiento de la red. Después, se comprueba su eficacia y su capacidad de aprendizaje y adaptación a casos nuevos verificando su comportamiento con los datos reservados para la validación.

Para implementar una red neuronal de estas características utilizando PyTorch, en el Código 2.7 hemos creado una clase heredada del objeto `Dataset` de PyTorch, en la cual almacenamos nuestros datos y sus respectivas clases a la vez, accediendo a los datos en formato de `tensor`, el tipo de dato de PyTorch optimizado para operaciones en GPU.

Para inicializar nuestros datos de entrenamiento y validación, en el Código 2.8 utilizamos la función `random_split`, que se encarga de distribuir nuestros puntos de forma aleatoria en los distintos subconjuntos. Además, fijamos el tamaño de los lotes que vamos a pasar a la red durante el entrenamiento, antes de actualizar los parámetros minimizando la función de pérdida, e inicializamos dos `DataLoader` que se encargaran de seleccionar aleatoriamente y en el tamaño de lote seleccionado, datos para pasar a la red tanto en el entrenamiento como en la validación.

Finalmente, en el Código 2.9 inicializamos nuestra red con dos capas lineales y una función de activación `Tanh` entre medias. La primera capa tiene 3 entradas, una por cada variable de los puntos artificiales y un número arbitrario de 128 salidas. Tras pasar por la función de activación, la segunda capa recibe esas 128

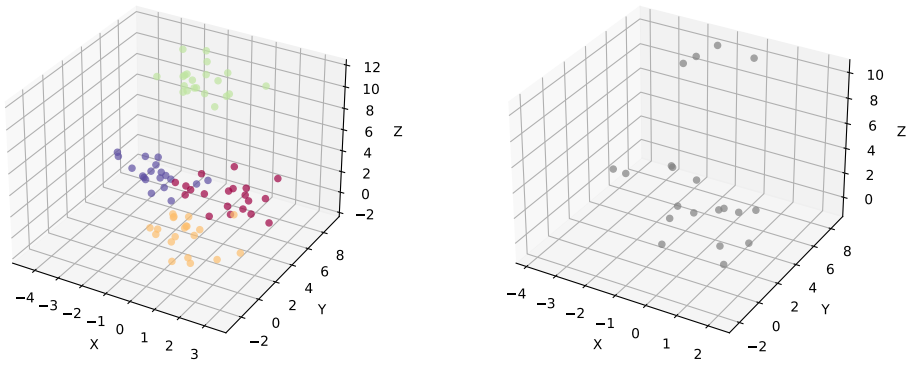
Código 2.7: `Dataset` de los datos artificiales.

```
from torchvision import torch
from torch.utils.data import Dataset

class RawDataset(Dataset):
def __init__(self, arr, labels):
self.data = arr
self.labels = labels

def __len__(self):
return len(self.data)

def __getitem__(self, idx):
data = self.data[idx]
label = self.labels[idx]
return torch.tensor(data, dtype=torch.float), label

dataset = RawDataset(X, y)
```

salidas de la capa anterior como sus entradas y saca 4 salidas, una por cada grupo que queremos distinguir.

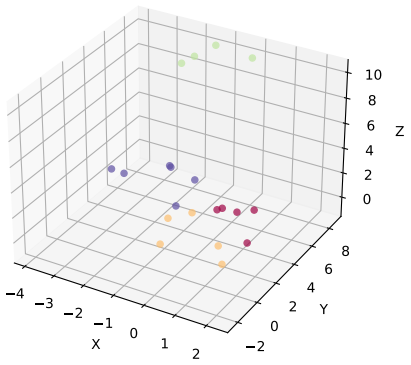
Posteriormente se escogen una función de pérdida y un optimizador de entre los proporcionados por PyTorch, y se fijan la tasa de aprendizaje y el número de iteraciones que se quieren realizar.

Una vez todos los componentes están listos, se itera en el bucle principal tantas veces como se desee ejecutando los pasos de entrenamiento explicados previamente. En la Figura 2.7 se observan los resultados de la validación, observando que esta red básica es suficiente para clasificar todos los dato de validación correctamente.



(a)

(b)



(c)

Figura 2.7: Prueba de clasificación mediante una red neuronal secuencial. 2.7a: Subconjunto de entrenamiento de los datos artificiales. Generado seleccionando 80 puntos de forma aleatoria manteniendo la información del grupo al que pertenecen para realizar el entrenamiento de nuestra red secuencial. 2.7b: Subconjunto de validación de los datos artificiales. Formado por los 20 puntos restantes que no se han seleccionado para el subconjunto de entrenamiento. Se guarda la información de sus grupos originales, pero no se le pasa a la red, para así obtener una clasificación automática y poder posteriormente verificar los resultados. 2.7c: Clasificación de los datos de validación generada por la red. Acierta el grupo real de los 20 puntos con los que se ha verificado su comportamiento.

Código 2.8: Inicialización de los conjuntos de entrenamiento y validación.

```
from torch.utils.data import DataLoader, random_split

train_size = int(0.8 * len(dataset))
test_size = len(dataset) - train_size

train_dataset, test_dataset = \
random_split(dataset, [train_size, test_size])

batch_size = 3
train_loader = DataLoader(train_dataset,
batch_size=batch_size,
shuffle=True)
test_loader = DataLoader(test_dataset,
batch_size=batch_size,
shuffle=False)
```

2.2.2. Red neuronal convolucional

Para resolver problemas simples como la clasificación de los puntos artificiales con los que hemos estado trabajando, las redes neuronales secuenciales lineales como la que hemos visto funcionan perfectamente, sin embargo, a la hora de analizar datos reales no siempre consiguen los resultados que nos gustaría. Para arreglar esto se puede valorar aumentar el número de capas de la red, aumentar el tamaño de las capas, probar diferentes funciones de activación, o aumentar el número de datos de entrenamiento. Otra posible opción es probar con redes que funcionen algo diferente como, por ejemplo, las redes neuronales convolucionales.

Las redes secuenciales relacionan cada punto del vector de entrada con todos los demás `asumiendo` que pueden existir relaciones entre cualquier par de puntos. Las redes convolucionales, en cambio, incorporan capas encargadas de aplicar una serie de filtros sobre los datos para asociar y dar relevancia a puntos próximos entre sí. Las redes convolucionales son ampliamente utilizadas para el análisis de imágenes, ya que es frecuente que las estructuras y características que podamos querer buscar en una imagen, estén formadas por píxeles próximos entre sí. En nuestro caso, vamos a analizar datos en series temporales, por lo que de la misma forma, es probable que las estructuras que queremos encontrar estén próximas temporalmente.

No vamos a centrarnos a fondo en los detalles matemáticos de estas redes, pero en la Sección 3.3 vamos a implementar una red convolucional a medida para nuestros datos de laboratorio utilizando PyTorch.

Código 2.9: Modelo de la red y enrenamiento con los datos artificiales.

```
import torch.nn as nn

model = nn.Sequential(
nn.Linear(3, 128),
nn.Tanh(),
nn.Linear(128, 4))
loss_fn = nn.CrossEntropyLoss()
learn_rate = 1e-2
optimizer = torch.optim.SGD(model.parameters(), lr=learn_rate
   )
n_epochs = 100

for epoch in range(n_epochs):
for sessions, labels in train_loader:
batch_size = sessions.shape[0]
outputs = model(sessions.view(batch_size, -1))

loss = loss_fn(outputs, labels)
optimizer.zero_grad()
loss.backward()
optimizer.step()
```

3

Análisis y resultados

Una vez adquiridos los conocimientos teóricos necesarios, y comprendido el funcionamiento de los diferentes algoritmos que vamos a utilizar, estamos en disposición de aplicarlos a los datos de experimentación y comprobar la eficacia de los algoritmos en un caso real. Este capítulo aborda el trabajo realizado durante tres meses en el Instituto Cajal, aplicando diferentes técnicas para tratar de agrupar los comportamientos de los animales, y de diferenciar animales bajo efectos de tratamiento o de placebo.

En la Sección 3.1 describimos brevemente las herramientas que hemos utilizado para realizar el trabajo. En la Sección 3.2 tratamos el proceso de como hemos preprocesado los vídeos para homogeneizar todas las sesiones de experimentación, obtener los datos de posición de los animales en ellas y filtrar e interpolar los datos poco verosímiles. En la Sección 3.3 desarrollamos el proceso de análisis de los datos preprocesados aplicando los diferentes algoritmos vistos anteriormente y analizando sus resultados.

3.1. Bibliotecas y herramientas utilizadas

En esta sección describimos brevemente las diferentes bibliotecas de Python que hemos usado para el desarrollo del código del trabajo, además de comentar el entorno de trabajo que hemos utilizado para poder realizar computación en la nube.

DeepLabCut

DeepLabCut [7] es una herramienta para la estimación de la posición de animales sin marcadores mediante el uso de redes neuronales. Es capaz de identificar de rastrear diferentes partes del cuerpo de múltiples especies realizando todo tipo de comportamientos. Esta ha sido la base de todo nuestro trabajo, ya que todos los vídeos que hemos analizado han sido procesados en primer lugar por DeepLabCut para rastrear las posiciones de múltiples puntos de los animales a lo largo de los vídeos de las sesiones. Debido a su importancia, damos una explicación más en detalle de su funcionamiento y de como lo hemos utilizado en la sección 3.2.1.

Google Colab

Google Colab es una herramienta para realizar cuadernos de Jupyter en línea, y poder ejecutarlos en el *backend* de Google. Estos son documentos que intercalan fragmentos de texto con fragmentos de código ejecutable en Python, así como la salida de las distintas ejecuciones. Todo el codigo realizado para este trabajo ha sido realizado en cuadernos de Colab, y puede ser consultado en los siguientes enlaces:

Ejemplos teóricos	https://colab.research.google.com/drive/1qL-LQTCFLZcqExN7qyYyRAT1nm7P8NhuK?usp=sharing
DeepLabCut	https://colab.research.google.com/drive/1dFb-mUcfW9el50v0RBCkLIItTD6SF7A3x?usp=sharing
Análisis principal	https://colab.research.google.com/drive/1ak2VpDi-zTnV-uEDp-viEkpa8GFDGuBR?usp=sharing

Tabla 3.1: Enlaces a cuadernos de Google Colab.

Scikit-learn

Scikit-learn [8] es una biblioteca de código abierto de Python que contiene numerosas implementaciones de algoritmos de aprendizaje automático. Hemos usado dichas implementaciones tanto como para la explicación teórica de los algoritmos, como para el análisis real de los datos de los animales.

Pandas

Pandas es una biblioteca de manejo de datos mediante tablas denominadas `DataFrame`. Todos los datos que hemos cargado para ser analizados los hemos

guardado como `DataFrames` para poder tener un fácil acceso a todos ellos y a sus variables pudiendo, además, visualizarlos de una forma sencilla.

NumPy

NumPy es una de las bibliotecas principales de cálculo científico en Python. Proporciona un objeto de `array` multidimensional y numerosas funciones estadísticas y algebraicas de mucha utilidad.

PyTorch

PyTorch [9] es una biblioteca de Python desarrollada por Facebook para la implementación de redes neuronales. Incluye multitud de módulos que facilitan la creación de redes neuronales a medida y varios optimizadores para acelerar el proceso de entrenamiento. Además, proporciona el objeto `Tensor` similar al `array` de NumPy, pero optimizado para el cómputo en procesadores gráficos. Hemos utilizado todas estas herramientas para implementar una red neuronal para analizar nuestros datos de laboratorio.

Matplotlib

Matplotlib es una biblioteca para crear figuras en Python. Todas las figuras de representación de datos que aparecen en este trabajo han sido creadas con `matplotlib`. El código concreto utilizado para generarlas puede consultarse en los cuadernos de Colab.

3.2. Preprocesado de datos

Los datos de los que hemos partido para realizar el análisis han sido 88 pares de vídeos en blanco y negro, de cinco minutos cada uno, de ratones en su caja hábitat sin estar realizando ninguna tarea concreta. Cada par de vídeos consistía en un video de la vista cenital de la caja y otro de la vista lateral, como se puede observar en la Figura 3.1. Además, de cada par de vídeos teníamos el código del animal que estaba siendo grabado, la fecha de la sesión y si el animal estaba bajo los efectos del tratamiento de NMDA en dicha fecha o no. Mostramos un resumen de estos datos en la Tabla 3.2.

Para tratar de averiguar de forma automática el tipo de tratamiento de cada sesión, podríamos entrenar una red neuronal utilizando directamente los vídeos

Video	Código del animal	Fecha de la sesión	Tratamiento
1	4128	2020-12-02	CONTROL
2	4128	2020-11-21	CONTROL
3	4128	2020-11-23	CONTROL
...	...	...	...
80	4108	2020-09-25	NMDA
81	4089	2020-09-18	NMDA
82	4089	2020-09-12	CONTROL
83	4089	2020-09-24	NMDA

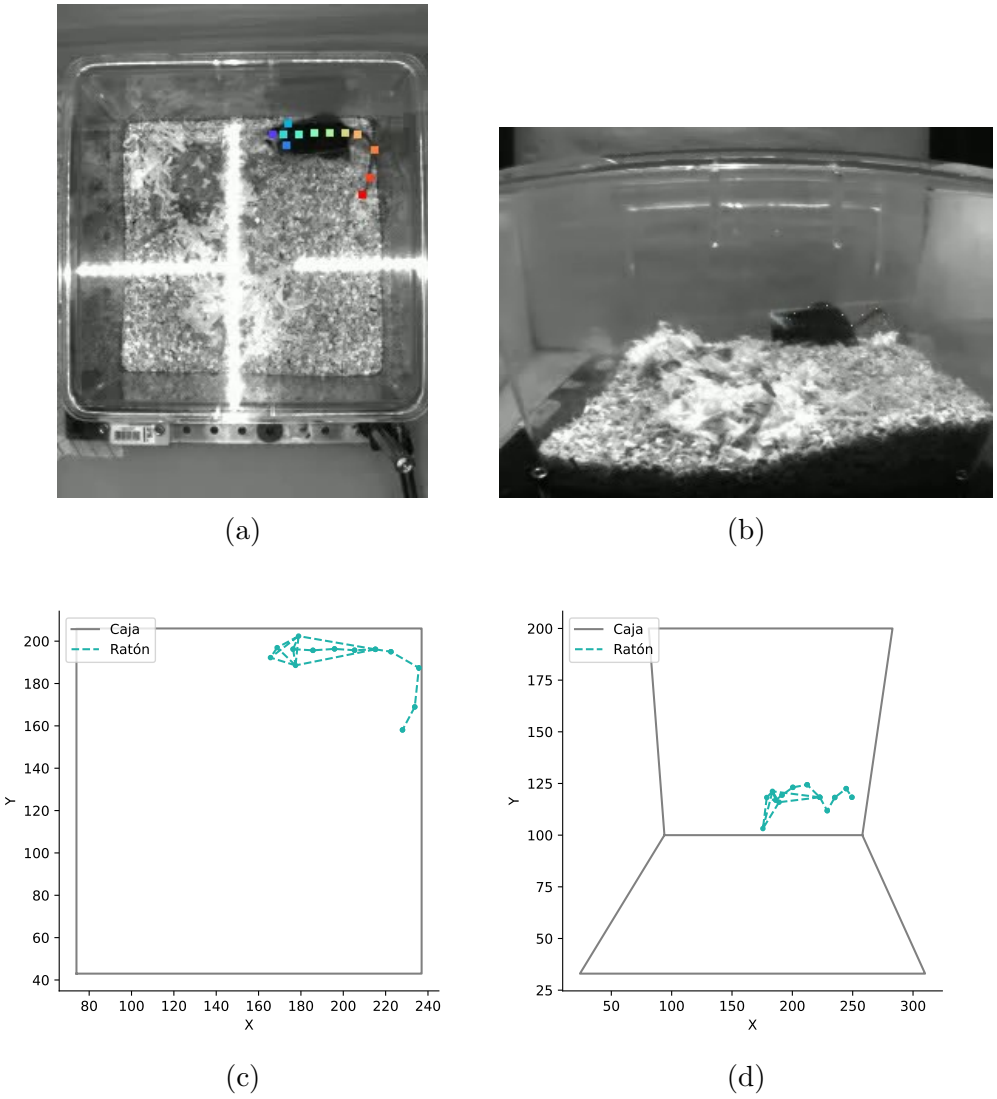
Tabla 3.2: Fragmento del `DataFrame` que almacena los datos de las sesiones, incluyendo el número del video, el código del animal, la fecha de la sesión y el estado del tratamiento.

como datos de entrenamiento. Sin embargo, no contamos con una cantidad muy elevada de vídeos para entrenar, y además esto no nos daría ninguna pista sobre como comportamientos concretos se relacionan con el tratamiento. Al no contar con una base de datos de fragmentos de vídeos clasificados según el comportamiento del animal, tampoco podemos entrenar una red para distinguir estos comportamientos. Por ello, hemos hecho uso de la herramienta DeepLabCut para rastrear la posición y la postura del ratón en cada fotograma y así poder hacer cálculos sobre ella, trabajar con otros algoritmos de agrupación no supervisada para clasificar diferentes movimientos del ratón a lo largo del video, o tener datos independientes del entorno con los que poder entrenar otras redes neuronales para distinguir el estado del tratamiento de cada animal.

3.2.1. DeepLabCut

DeepLabCut es una red convolucional profunda que combina redes residuales pre entrenadas con capas deconvolucionales. Una red residual es aquella que concatena cientos de capas convolucionales con capas deconvolucionales que amplían la información visual para producir densidades de probabilidad espaciales. Esto hace que ajustar DeepLabCut para realizar una tarea concreta conlleve únicamente entrenar una red utilizando pocos datos de entrenamiento para obtener un listado de coordenadas de los puntos deseados acompañadas de la verosimilitud de cada uno de los puntos predichos. Las capas deconvolucionales están pre entrenadas con la base de datos de ImageNet [10], una base de datos de imágenes organizadas en conjuntos de sinónimos cognitivos, cada uno agrupando un concepto diferente. El objetivo de ImageNet es proporcionar una buena base de entrenamiento para investigaciones de todo tipo, como por ejemplo la de DeepLabCut. La Figura 3.2 muestra un esquema de la estructura de la red.

En el código 3.1 podemos observar como hemos inicializado nuestros conjun-



(a) (b)

(c) (d)

Figura 3.1: Salida de DeepLabCut del animal 4128 el 02-12-2020, 1:00:37. 3.1a Video de la vista cenital de la caja. Los puntos sobre el animal son los dibujados por DeepLabCut para rastrear las partes del animal. 3.1b Video de la vista lateral de la caja. 3.1c En azul, la triangulación formada por los puntos obtenidos por DeepLabCut de la vista cenital. Cinco puntos para la cabeza, cuatro para la espalda y otros cuatro para la cola. En gris, las dimensiones de la base de la caja, obtenidas midiendo las distancias en píxeles sobre un fotograma de video. 3.1d Análogo a la vista cenital, desde la vista lateral. En gris, la altura mínima del suelo de la caja y su pared trasera.

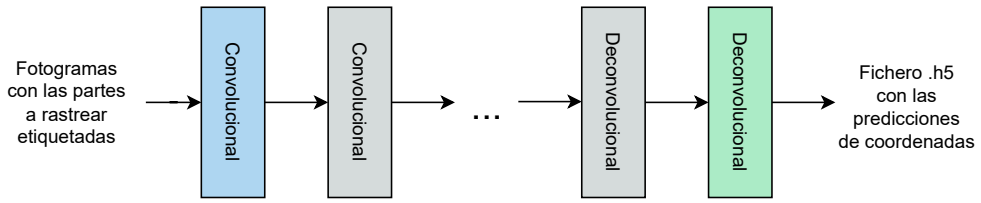


Figura 3.2: Diagrama simplificado de la red utilizada por DeepLabCut. La red recibe como entrada los datos de los fotogramas con las partes etiquetadas y los procesa por múltiples capas convolucionales pre entrenadas con ImageNet. Finalmente, las salidas de esas capas pasan por capas deconvolucionales para obtener el fichero con las predicciones de coordenadas.

tos de datos de entrenamiento para procesar los vídeos de las sesiones. Hemos realizado este proceso para los vídeos cenitales y laterales independientemente, indicando para cada uno de ellos los puntos que queremos rastrear en el fichero de configuración de `path_config_file`. Hemos utilizado también la red residual de 50 capas `resnet_50` pre-entrenada por DeepLabCut y seleccionado el método de aumentación de los datos de las capas deconvolucionales `imgaug`.

Código 3.1: Crear datos de entrenamiento de DeepLabCut.

```
import deeplabcut

deeplabcut.create_training_dataset(path_config_file,
                                   net_type='resnet_50',
                                   augmenter_type='imgaug')
```

Tras crear los datos de entrenamiento, en el laboratorio se habían etiquetado manualmente 10 fotogramas por sesión, los cuales hemos utilizado en el código 3.2 para entrenar la red.

Código 3.2: Entrenar la red de DeepLabCut.

```
deeplabcut.train_network(path_config_file,
                         shuffle=1,
                         trainingsetindex=0,
                         max_snapshots_to_keep=5,
                         displayiters=10,
                         saveiters=250,
                         maxiters=1000,
                         allow_growth=False,
                         autotune=False,
                         keepdeconvweights=True)
```

Tras entrenar la red la evaluamos para verificar que tenemos un error aceptable con el Código 3.3.

Código 3.3: Evaluar la red de DeepLabCut.

```
deeplabcut.evaluate_network(path_config_file)
```

Tras entrenar la red, analizamos los vídeos con `deeplabcut.analyze_vídeos` obteniendo un fichero `h5` para cada una de las sesiónes con las cordenadas de todos los puntos rastreados y la verosimilitud de cada uno de los puntos. En la Tabla 3.3 mostramos un fragmento de uno de los ficheros cargado a memoria en un `DataFrame` para su posterior preprocesado y análisis. Cada uno de los ficheros contiene los datos de 13 partes del animal rastreadas en cada uno de los fotogramas. Las 13 partes se han denominado `Nose`, `Head`, `Neck`, `Leftear`, `Rightear`, `Back{1-4}`, `Tail{1-4}`. Por cada una de las partes, DeepLabCut nos proporciona su predición para las coordenadas x e y y la verosimilitud de dicha predicción.

Finalmente, utilizando la función `deeplabcut.create_labeled_video` hemos creado dos vídeos de muestra con los puntos rastreados dibujados sobre cada uno de los fotogramas del video como se puede observar en la Figura 3.1.

	Nosex	Nosey	Noselikelihood	Headx	Heady	...
0	136.165344	177.722496	0.000084	129.790253	174.772552	...
1	162.032005	201.444756	0.942181	168.152061	202.420639	...
2	156.297043	200.326378	0.000073	162.436028	203.156837	...
3	155.370415	199.043297	0.000277	159.507599	200.199928	...
4	149.272644	197.677170	0.000045	155.493912	198.814835	...
...	...	...	...	...	...	...

Tabla 3.3: Extracto del `DataFrame` de Pandas de los datos sin procesar de DeepLabCut. Para cada punto rastreado, y para cada fotograma del video, se computa la predicción de la coordenada x y de la coordenada y y se da la verosimilitud de dicha predicción.

3.2.2. Filtrado e interpolación

Una vez hemos pasado todos los vídeos por DeepLabCut, tenemos dos ficheros de datos para cada uno de las sesiones, uno para las coordenadas de los puntos de la vista cenital y otro para las coordenadas de los puntos de la vista lateral. Al cargar los ficheros a memoria como `DataFrames` de `pandas`, observamos que no todos ellos tienen la misma duración, por lo que los homogeneizamos todos eliminando los datos de los últimos fotogramas, consiguiendo que cada `DataFrame` tenga un tamaño de 6000 filas, lo que equivale a 6000 fotogramas o 5 minutos de video. Además, eliminamos los datos de 4 vídeos que no llegan a esa duración, ya que son vídeos con problemas que entorpecerían el análisis.

Por cada par de coordenadas de cada parte del animal en cada fotograma, DeepLabCut estima la precisión de su predicción y asocia un valor de verosimilitud del par. Antes de trabajar con la salida de DeepLabCut filtramos todos los pares de coordenadas para quedarnos solo con aquellos que tienen una verosimilitud de más de 0,95. Tras el filtrado, interpolamos linealmente las coordenadas eliminadas basándonos en las predicciones de la misma parte más próximas en tiempo con verosimilitud suficiente. En la Figura 3.3 se observa la evolución de los datos de las trayectorias a lo largo del proceso de filtrado e interpolación.

Finalmente, eliminamos los datos de 2 vídeos más, ya que tras el preprocesado seguían teniendo valores nulos debido a que DeepLabCut no ha podido identificar varios de los puntos a rastrear en ninguno de los fotogramas de los vídeos.

3.3. Análisis de datos

En esta sección desarrollamos los diferentes métodos que hemos utilizado en nuestro análisis de los datos, junto con los resultados que hemos ido obteniendo. Primero explicamos los diferentes cómputos manuales que hemos realizado con nuestros datos para identificar ciertos comportamientos y posteriormente explicamos los algoritmos de aprendizaje automático que hemos utilizado para realizar distintas clasificaciones. Para realizar cada uno de los métodos, partimos de los datos de 81 sesiones de video preproesadas. Cada una de las sesiones cuenta con 6000 filas, una por fotograma, y 52 coordenadas, coordenadas x e y por cada una de las 13 partes rastreadas por cada una de las dos vistas, cenital y lateral.

3.3.1. Clasificación manual de comportamientos

Gracias a los datos de DeepLabCut podemos tratar de identificar ciertos comportamientos de forma manual, especificando ciertos umbrales para extraer los comportamientos deseados. A modo de ejemplo, hemos calculado cuando el punto de la cabeza en la vista cenital, `head`, se salía durante un intervalo de cinco fotogramas fuera de los límites establecidos de la caja. Así, como se observa en la Figura 3.4, hemos identificado los intervalos de tiempo de cada sesión en los cuales el animal se alza sobre sus extremidades traseras apoyándose en la pared de la caja.

De la misma forma, podemos calcular en que intervalos la media de los puntos de la espalda tienen una velocidad mayor de tres píxeles por segundo para identificar cuando el animal se está moviendo como se puede ver en la Figura 3.5, o calcular el ángulo formado entre un vector de los puntos de la cabeza del animal y otro de los puntos de la espalda, para determinar si el animal está girado hacia algún lado, visualizado en la Figura 3.6.

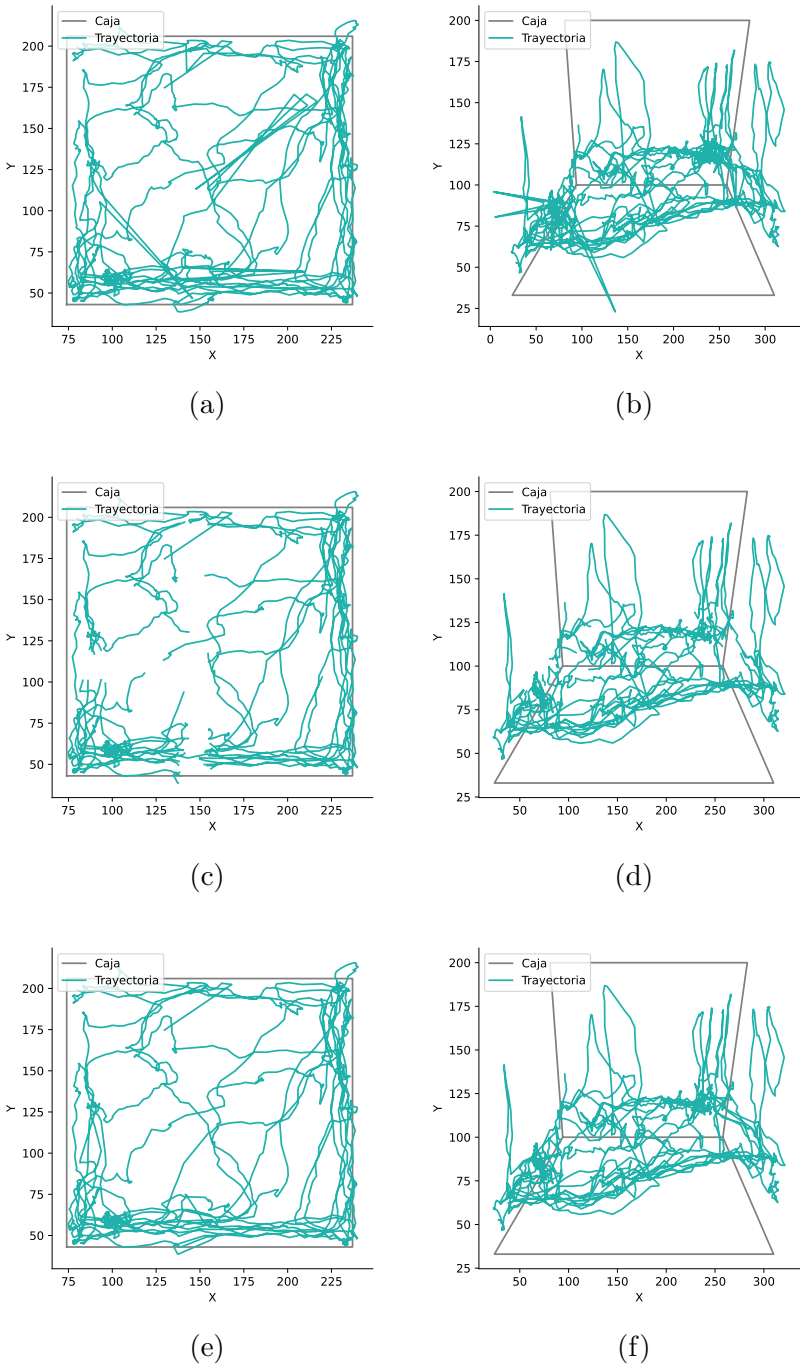


Figura 3.3: Vistas cenital y lateral de trayectorias de un punto de la cabeza del animal 4128. Datos correspondientes a los dos primeros minutos de la sesión del 2020-12-02. 3.3a, 3.3b: Vista cenital de los datos antes del preprocesado. En gris se ha dibujado las dimensiones de la caja y en color la trayectoria del punto `Head`. 3.3e, 3.3f: Vistas cenital y lateral de los datos una vez filtrados los valores de baja verosimilitud. 3.3e, 3.3f: Vistas cenital y lateral de los datos filtrados con los valores nulos interpolados.

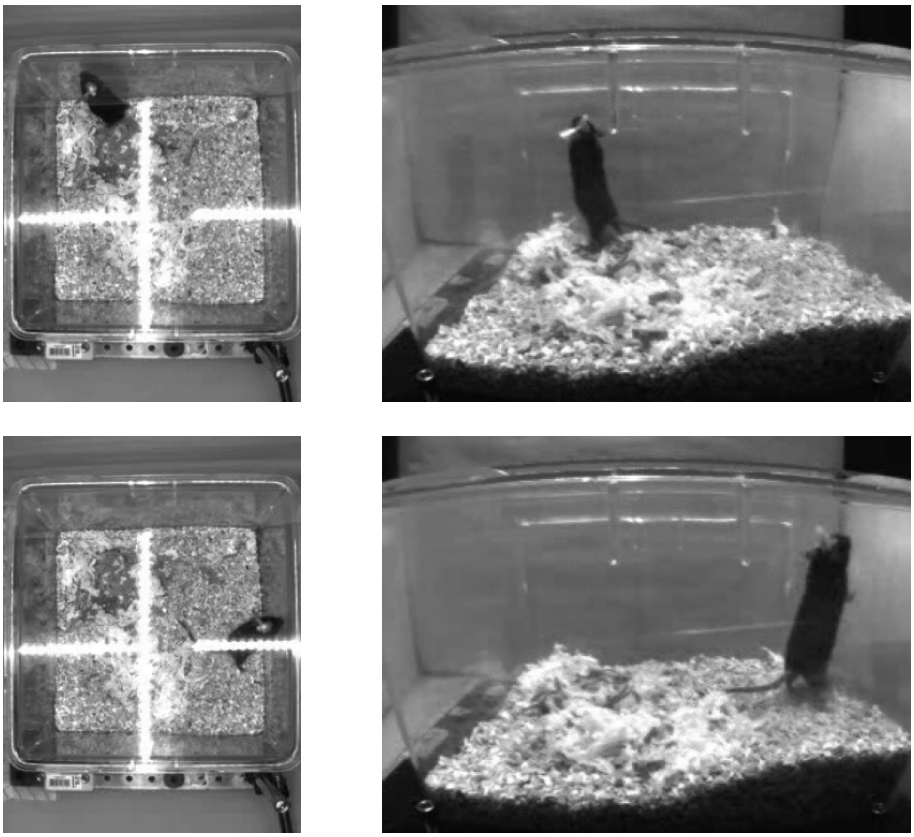


Figura 3.4: Fotogramas de los dos primeros intervalos en los que el animal 4128 se alza en la sesión del 2020-12-02. Los fotogramas superiores corresponden a la vista cenital y lateral del segundo 3,05. Los fotogramas inferiores corresponden a las mismas vistas del segundo 19,43.

Estos métodos de extracción de comportamientos funcionan para este caso particular, pero al depender de funciones hechas específicamente para estos datos, sería complejo exportarlo a otros escenarios. DeepLabCut ayuda en ese aspecto, ya que no calculamos propiedades sobre los vídeos en sí, sino sobre un conjunto de coordenadas de posiciones de partes del animal. De esta forma, si se tratase de analizar vídeos de animales grabados en otras cajas o realizando alguna tarea, podríamos computar de forma sencilla estas propiedades ajustando los parámetros necesarios.

Sin embargo, para el cómputo manual de estas propiedades se han puesto umbrales arbitrarios para definir el estado del animal. Un umbral de 3 píxeles por segundo sobre la media de los puntos de la espalda para determinar cuando el animal se está desplazando, un umbral de $\pm 30^{\circ}$ para determinar si el animal está girando o no, o un umbral de 5 fotogramas fuera de las dimensiones de la caja para determinar si el animal se está alzando sobre una pared. Los umbrales son necesarios si consideramos necesario discretizar estas variables continuas en

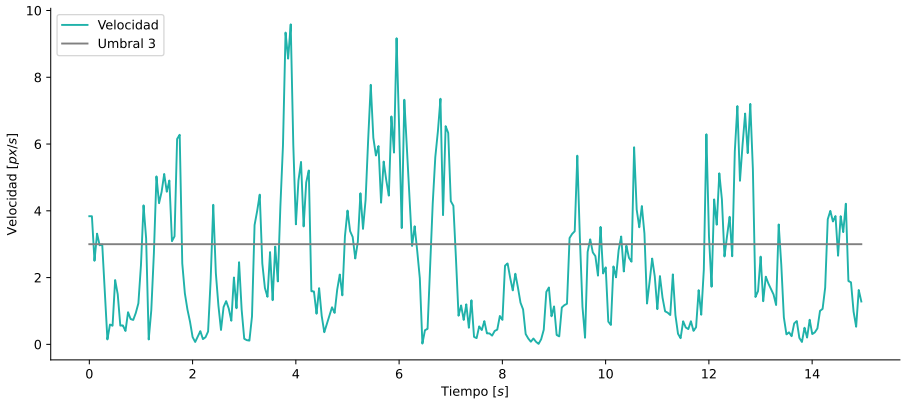


Figura 3.5: Gráfico de la velocidad del animal 4128, durante los primeros 15 segundos de la sesión del 2020-12-02. En color, los datos obtenidos derivando la trayectoria de la media de los puntos de la espalda del animal: `Neck`, `Back_1`, `Back_2`, `Back_3` y `Back_4`. En gris, el umbral arbitrario de 3 píxeles por segundo, para diferenciar cuando el animal está desplazándose y cuando no.

comportamientos cualitativos, pero están a su vez sesgados por el juicio del experimentador, ya que una varianza en la magnitud de los umbrales o en el conjunto de puntos utilizados para calcularlos puede resultar en una discrepancia de la clasificación obtenida.

La clasificación manual hace que sea complejo determinar también cuando el animal realiza algún comportamiento más complejo que los vistos hasta ahora, ya que si no sabemos de antemano la relación entre las coordenadas de las partes del animal y el comportamiento, no podremos identificarlo. De la misma forma, al no conocer de antemano la relación entre los diferentes comportamientos y el tipo de tratamiento dado a los animales, con estas clasificaciones no se podrán diferenciar animales tratados con anticuerpos de NMDA de animales control.

Esto motiva las siguientes secciones en las cuales hemos utilizado diversos métodos de aprendizaje automático para tratar de conseguir clasificaciones sin depender de sesgos preestablecidos.

3.3.2. Agrupaciones no supervisadas

Para tratar de automatizar el proceso de clasificación, sin depender de cualidades computadas *ad hoc* para este conjunto de datos, hemos aplicado los algoritmos vistos en la sección 2.1 sobre los datos de las trayectorias preprocesados según hemos visto en la sección 3.2.

Para clasificar los comportamientos de un animal en cada sesión no conta-

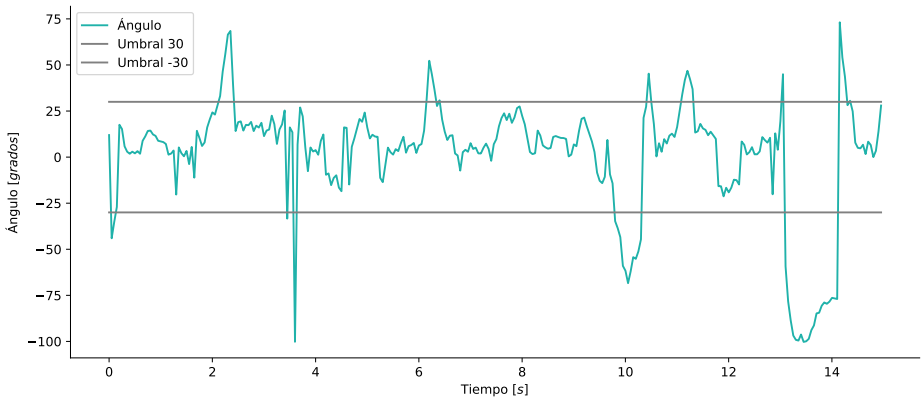
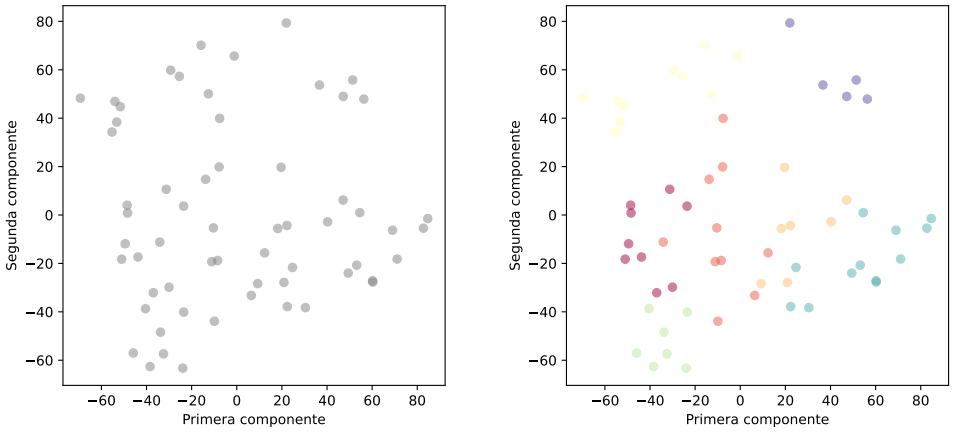


Figura 3.6: Gráfico de los ángulos del animal 4128, durante los primeros 15 segundos de la sesión del 2020-12-02. En color, el ángulo formado por el vector de los puntos de la cabeza con el vector de los puntos de la espalda. En gris, los umbrales de 30° y -30° para determinar si el animal está recto, o tiene el cuerpo girado hacia la izquierda o derecha respectivamente,

mos ejemplos de comportamientos identificados y previamente clasificados por lo que no podemos hacer uso de métodos de aprendizaje automático supervisados. Además, si no queremos introducir un sesgo en la clasificación, tampoco sabemos de antemano en cuantos grupos de comportamiento similares podremos agrupar los diferentes fragmentos de cada sesión de video. Por ello hemos utilizado un algoritmo de propagación de afinidad, basándonos en el trabajo realizado por A. Klaus et al. [11]. Dicho algoritmo nos permite agrupar fragmentos de comportamiento función de su afinidad. Hemos dividido los vídeos de las sesiones en intervalos de 5 segundos e introducido los datos de las coordenadas de todos los puntos de las vistas cenital y lateral para todos los puntos de cada fotograma de los intervalos. En la Figura 3.7 se muestra la agrupación obtenida para la sesión de uno de los animales, consiguiendo agrupar los distintos intervalos en 7 grupos de comportamiento.

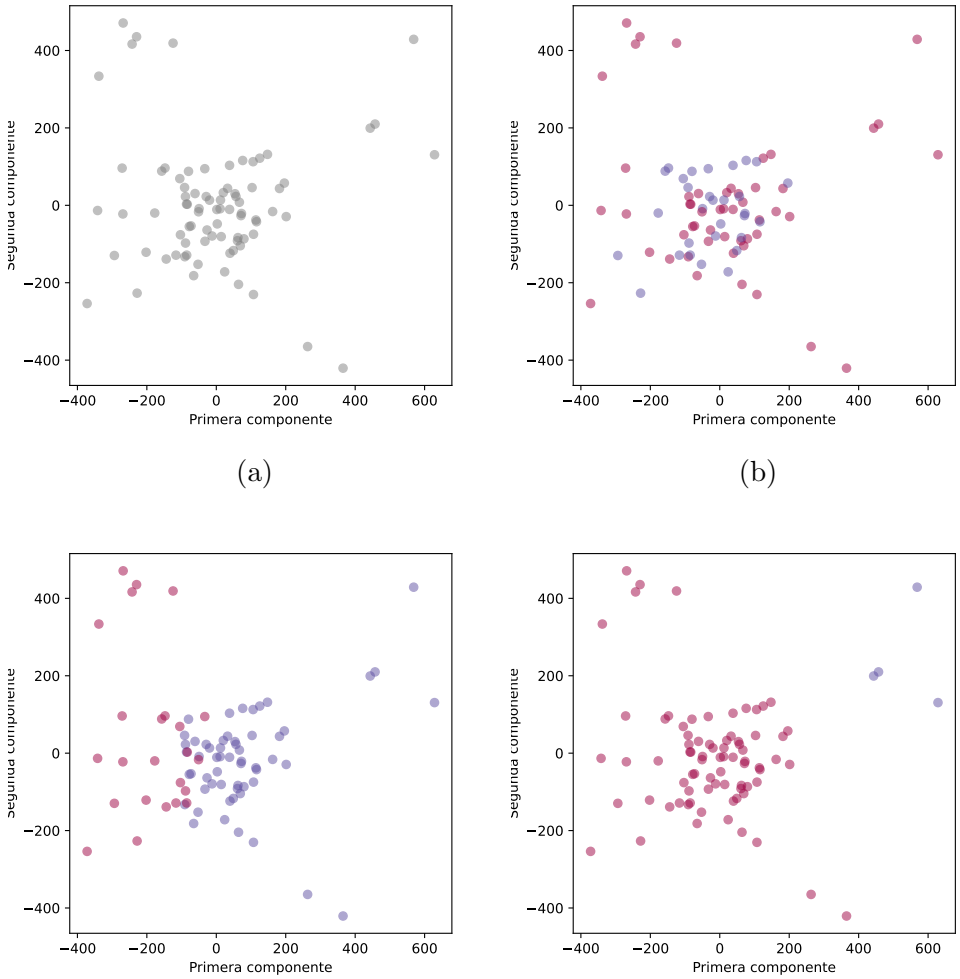
No podemos usar el algoritmo de propagación de afinidad para diferenciar el tipo de tratamiento de cada una de las sesiones, ya que no es posible garantizar que vaya a generar dos únicas clases. Por ello, hemos utilizado los otros dos algoritmos explicados en la sección 2.1: la agrupación por k-medias y la agrupación aglomerativa, ambos utilizados en estudios similares [12]. Para ambos casos hemos tomado los datos de todas las sesiones y tras aplicarles el algoritmo hemos mostrado los resultados en la Figura 3.8. Con ninguno de los dos algoritmos hemos conseguido resultados mejores que los que obtendríamos haciendo asignaciones aleatorias de las clases, por lo que no podemos asegurar que exista alguna relación directa entre la posición de los animales y el tipo de tratamiento que se les ha suministrado.



(a) (b)

Figura 3.7: Clasificación de comportamiento mediante el algoritmo de propagación de afinidad sobre intervalos de la sesión del animal 4128 el 2020-12-02. 3.7a: Distribución de los puntos de un PCA sobre una subdivisión de la sesión en 60 intervalos. Cada punto representa un intervalo de 5 segundos, 100 fotogramas, de la sesión de video. Cada punto original cuenta con 56 dimensiones, correspondientes a las coordenadas x e y de los 13 puntos extraídos por DeepLabCut tanto para la vista cenital como la vista lateral. Todos los puntos se han normalizado para tener media 0 y desviación 1 antes de realizar el PCA. Tras el PCA, y únicamente para su representación gráfica, se conservan las dos primeras componentes principales de cada punto. 3.7b: Misma representación de los 60 intervalos que en 3.7a, pero los puntos han sido coloreados según el número de grupo asignado por el algoritmo de propagación de afinidad. El algoritmo se ha realizado sobre los puntos de los intervalos normalizados, manteniendo las 56 dimensiones originales y utilizando la norma euclídea como medida de afinidad.

Para conseguir realizar esta clasificación habría que computar otras características manualmente con las que dar más información a los algoritmos, cambiar los puntos rastreados por DeepLabCut para conseguir información sobre otras partes del animal, o tratar de predecir una función desconocida que fuera capaz de agrupar las sesiones correctamente. Como tenemos los datos de todas las sesiones podemos entrenar métodos supervisados, y estamos en las condiciones idóneas para tratar de predecir, en caso de que exista, esa función que relacione las coordenadas de los puntos que hemos rastreado con el tipo de tratamiento de cada sesión.



(a) (b)

(c) (d)

Figura 3.8: Clasificaciones no supervisadas del tipo de tratamiento. 3.8a: Distribución de los puntos de un PCA realizado sobre los datos de las 81 sesiones preprocesados y normalizados. Cada punto tiene una dimensión original de 6000 fotogramas por 52 coordenadas por fotograma, que se han linealizado para tener 312000 dimensiones por punto antes de realizar el PCA. 3.8b: Asignación real del estado de cada tratamiento para cada sesión. En azul se han marcado las sesiones bajo el efecto de anticuerpos de NMDA y en rosa se han marcado las sesiones en las que no hay efecto, o se ha suministrado un tratamiento placebo. 3.8c: Resultados de una agrupación mediante un algoritmo de k-medias sobre los puntos de las 81 sesiones con todas las dimensiones, mostrado sobre los puntos obtenidos mediante el PCA. Clases agrupadas con una precisión del $51,85\,\%$. 3.8d Resultados de una agrupación mediante un algoritmo de agrupación aglomerada sobre los puntos de las 81 sesiones con todas las dimensiones, mostrado sobre los puntos obtenidos mediante el PCA. Clases agrupadas con una precisión del $58,02\,\%$.

3.3.3. Agrupaciones supervisadas

Para aproximar una función que relacione las coordenadas de los puntos rastreados por DeepLabCut con el estado de los animales en cada sesión de video, hemos realizado una red neuronal con dos capas convolucionales para relacionar propiedades comunes de fotogramas próximos en el tiempo, y con otras dos capas lineales para asociar las propiedades encontradas con una de las dos clases que queremos diferenciar: animales con tratamiento control o previos al tratamiento con anticuerpos anti NMDA, y animales que están bajo los efectos del tratamiento de anticuerpos anti NMDA.

Las redes convolucionales son usadas generalmente para el tratamiento de imágenes haciendo convoluciones bidimensionales, como por ejemplo en las redes de DeepLabCut. Como nuestros datos tienen una estructura de serie temporal, hemos realizado convoluciones unidimensionales con las capas `nn.Conv1d` de PyTorch. En datos de imágenes la convolución obtiene propiedades de píxeles próximos, en las series temporales se obtienen propiedades de momentos temporales próximos, fotogramas en nuestro caso.

Para preparar nuestros datos para entrenar y validar nuestra red hemos creado una subclase del tipo `Dataset` de PyTorch en la cual introducimos los `DataFrame` con los datos de las sesiones y las etiquetas de cada una de ellas para almacenarlas juntas y tener asociados los datos cada sesión con su etiqueta correspondiente. Podemos ver la implementación de esta clase en el Código 3.4. Al almacenar los datos también los normalizamos con el `StandardScaler` de Scikit-learn para facilitar la optimización de las funciones de la red. Además, hemos subdividido cada sesión de 5 minutos en fragmentos de un minuto para ampliar la cantidad de nuestros datos de prueba. Esto supone pasar de tener sesiones de 6000 fotogramas a sesiones de 1200, reduciendo a su vez la dimensión de cada uno de los puntos. Por contra, esto supone asumir que el periodo de tiempo de un minuto es suficiente para detectar patrones que puedan indicar el estado del tratamiento de un animal.

En el Código 3.5 podemos ver la implementación en PyTorch de la red que hemos diseñado, y en la Figura 3.9 un esquema con la estructura de la misma. Cuenta con un total de cuatro capas, dos convolucionales y dos lineales. La primera capa convolucional cuenta con una 52 canales de entrada, uno por cada coordenada de un fotograma. Su núcleo de tamaño 20 hace que al recorrer los 1200 fotogramas se asocien como próximos en grupos de 20, buscando estructuras de comportamiento de hasta un segundo. La capa tiene 104 salidas, duplicando el tamaño de los datos antes de introducirlos en la siguiente capa. Hemos utilizado una función de activación `relu` siguiendo los estándares de la mayoría a de redes de clasificación y hemos reducido la dimensionalidad de los datos en 2 antes de pasar los datos a la siguiente capa mediante la función `max_pool_1d`.

La segunda capa de nuestra red es otra convolucional que ahora reduce el número de canales de 104 a 26 con el mismo núcleo que la capa anterior, y volvien-

Código 3.4: `Dataset` de las sesiones

```
class SessionDataset(Dataset):
  def __init__(self, dataframes, labels):
    self.data = dataframes
    self.labels = np.array(labels)

    # Fit the scaler on the entire dataset
    self.scaler = StandardScaler()
    self.scaler.fit(np.concatenate(self.data))

  def __len__(self):
    return len(self.data)

  def __getitem__(self, idx):
    # Convert dataframe to numpy array
    data = self.data[idx].values

    # Normalize the data
    data = self.scaler.transform(data)

    label = self.labels[idx]
    data_tensor = torch.tensor(data, dtype=torch.float)
    return data_tensor.permute(1, 0), label

dataset = SessionDataset(split_dfs,
                         split_animal_info['real-assigment'])
```

do a usar la misma función de activación y de reducción de dimensionalidad. Tras aplicar estas dos capas convolucionales nos quedan datos de 26 canales respecto a los 52 originales y 300 puntos, respecto a los 1200 con los que empezamos.

Finalmente aplicamos dos capas lineales como en las redes secuenciales tradicionales para transformar la salida de las capas convolucionales primero de 26 * 300 dimensiones a 32, y finalmente de esas 32 a 2 para determinar, para cada minuto de sesión, una de las dos clasificaciones que buscamos.

Para entrenar y validar nuestra red, obteniendo una medida estadísticamente robusta de la precisión de la misma, hemos utilizado un método de validación cruzada, separando nuestros datos en cinco grupos aleatorizados con un 20 % de los fragmentos de sesiones cada uno. Así, hemos entrenado cinco modelos de la misma red, cada uno con un 80 % de los datos, dejando en cada caso un 20 % de los datos diferente para realizar la validación. En el Apéndice A.1 hemos trascrito la salida del entrenamiento de los diferentes modelos observando que se obtienen precisiones que oscilan entre el 46 y 71 % de tasa de acierto, con una media final del 58,27 %, un porcentaje no muy superior al de realizar una asignación de clases

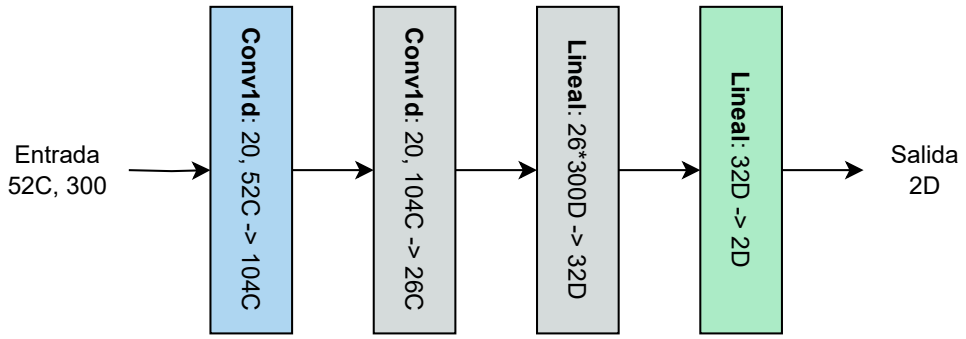


Figura 3.9: Diagrama de la red utilizada para analizar nuestros datos. La red recibe como entrada datos con 300 instantes temporales de 52 canales cada uno. Estos pasan a la siguiente capa duplicando los canales y a la tercera redudiéndose a 26. En las capas convolucionales, se usa un núcleo de tamaño 20, por lo que cada dato procesado solo tiene encuentra intervalos de 20 fotogramas próximos entre sí.

aleatoria.

Esta baja precisión puede estar causada por diversos factores, entre los cuales pueden estar la poca cantidad de datos de entrenamiento, el uso de una red demasiado genérica y poco adaptada al problema, o la posibilidad de que no exista una relación entre las coordenadas de los puntos rastreados de las partes del animal y el estado del tratamiento del mismo.

Código 3.5: Red convolucional

```
import torch.nn.functional as F

class TimeSeriesNet(nn.Module):
  def __init__(self):
    super().__init__()
    self.conv1 = nn.Conv1d(in_channels=52, out_channels=104,
                           kernel_size=20, padding=10)
    self.conv2 = nn.Conv1d(in_channels=104, out_channels=26,
                           kernel_size=20, padding=10)
    self.fc1 = nn.Linear(26 * 300, 32)
    self.fc2 = nn.Linear(32, 2)

  def forward(self, x):
    # Input shape: (batch_size, channels, sequence_length)

    # Max pooling with kernel size 2
    out = F.max_pool1d(F.relu(self.conv1(x)), 2)
    out = F.max_pool1d(F.relu(self.conv2(out)), 2)

    # Flatten for fully connected layer
    out = out.view(out.size(0), -1)
    out = F.relu(self.fc1(out))
    out = self.fc2(out)
    return out
```

4

Conclusiones

Los algoritmos de aprendizaje automático son una poderosa herramienta para analizar todo tipo de datos para multitud de disciplinas diferentes. En este trabajo hemos hecho una revisión de algunos de estos algoritmos explicando su funcionamiento y aplicándolos a un caso de uso en neurociencia de sistemas.

Hemos visto los dos usos principales de los algoritmos de aprendizaje no supervisados: la reducción de dimensionalidad de los datos y la agrupación de estos sin contar con una base de datos etiquetados previamente. Posteriormente, hemos introducido también los algoritmos de aprendizaje supervisado, dando la base matemática de las redes neuronales secuenciales y comentando las cualidades de las redes convolucionales.

El objetivo de este trabajo era usar este tipo de algoritmos para, dada la postura corporal de un ratón a lo largo de una sesión de video, diferenciar automáticamente diferentes comportamientos e identificar si un animal está bajo los efectos de un bloqueador de NMDA. Para ello, hemos partido de los datos de videos de sesiones procesados por DeepLabCut, obteniendo la postura corporal de cada ratón en cada uno de los fotogramas de video. A partir de estos datos, hemos utilizado tres técnicas de análisis diferentes: una clasificación manual de comportamientos, agrupaciones supervisadas y agrupaciones no supervisadas.

La clasificación manual ha funcionado para extraer ciertos comportamientos claramente diferenciables con los puntos de la postura corporal que hemos rastreado. Gracias a ella hemos identificado cuando el animal estaba en movimiento y cuando en reposo, el ángulo que formaba el animal en cada instante, y cuando se ponía a dos patas apoyándose sobre la caja. Pese a funcionar en estos casos, la

clasificación manual es un método poco escalable para identificar comportamientos más sofisticados, ya que para cada nuevo comportamiento a identificar hay que pensar en un algoritmo concreto que pueda diferenciar dicho comportamiento de otros. Además, no hemos sido capaces de utilizar estos algoritmos para diferenciar sesiones de animales tratados de animales no tratados, ya que no conocemos las características visuales que diferencian unos de otros.

Este tipo de algoritmos se pueden enmarcar bajo el término de inteligencia artificial rudimentaria, tomando decisiones en base a condiciones e instrucciones previamente codificadas para realizar esa tarea concreta. Para salvar estos problemas, hemos utilizado algoritmos de aprendizaje automático, otro subconjunto de algoritmos de inteligencia artificial, los cuales analizan multitud de datos para tomar decisiones de forma autónoma.

Primero hemos utilizado algoritmos de aprendizaje no supervisado, los cuales nos permiten prescindir de datos de entrenamiento, facilitando su aplicación. Para diferenciar el tipo de tratamiento hemos utilizado los algoritmos de k-medias y de agrupación aglomerativa, consiguiendo una tasa de aciertos en la clasificación de nuestros datos del 51,85 % y 58,02 % respectivamente. Ninguno de los dos resultados es muy satisfactorio, lo que puede deberse a varios factores. En ambas agrupaciones hemos utilizado la distancia euclídea como medida de similitud entre los puntos multidimensionales, sin embargo, quizás no sea la medida más apropiada a la luz de los resultados.

El algoritmo de k-medias y de agrupación aglomerativa necesitan de antemano el número de grupos que se desea realizar. Como para analizar los comportamientos no sabemos cuantos motivos comportamentales diferentes realiza el animal, podríamos haber realizado los algoritmos con diferente número de grupos y comprobar con el grado de cohesión de los grupos para cada agrupación. Sin embargo, hemos utilizado el algoritmo de agrupación por afinidad, que determina autónomamente cuantos grupos realizar. El problema que ha surgido de la utilización de este algoritmo es que es complejo verificar su funcionamiento. No ha generado el mismo número de grupos para todos los animales, y tampoco ha formado algún patrón que hayamos sido capaces de reconocer entre los diferentes tipos de sesiones. A fin de cuentas, ese es el problema al que se enfrentan los algoritmos no supervisados: realizan clasificaciones de los datos, pero no siempre el tipo de clasificación que estábamos buscando.

Finalmente, hemos utilizado técnicas de aprendizaje automático supervisado diseñado una red neuronal convolucional para diferenciar los tipos de tratamiento. No podemos utilizar esta técnica para diferenciar tipos de comportamientos porque no contamos con una base de datos de comportamientos etiquetados que nos sirvan para entrenar a nuestra red. Tras un proceso de validación cruzada, nuestra red ha obtenido una tasa de aciertos media del 58,27 %, lejos de ser un resultado reseñable. Esto puede ser debido a muchos factores. Al igual que para el análisis con algoritmos no supervisados, la elección de los puntos corporales a

rastrear puede no ser la más apropiada, a lo que hay que sumar que hemos contado con una base de datos relativamente pequeña en comparación a la cantidad de datos que suelen utilizarse para entrenar redes en otras disciplinas. También puede ser que la configuración de capas y entradas de nuestra red no sea la más apropiada para el problema, que las funciones de activación no sean las más indicadas, o que directamente el tipo de red elegida no sea la más adecuada para el problema.

A título personal, como autor de este trabajo, su realización me ha servido para ver como se trabaja en un entorno de laboratorio real, con científicos titulares, y otros estudiantes en prácticas o doctorando. Me ha ayudado a agilizar mi capacidad de aprendizaje autónoma, dándome la oportunidad de leer multitud de artículos científicos en diversos ámbitos de la Neurociencia de Sistemas y libros sobre Inteligencia Artificial y Aprendizaje Automático. Una de las partes más desafiantes del trabajo ha sido implementar una versión funcional de una red neuronal en PyTorch, resultándome, sin embargo, y pese a sus resultados finales, una de las partes más satisfactorias del trabajo.

Trabajos futuros pueden tratar de probar otras configuraciones de red neuronal, variando los parámetros y su composición. Además, pueden entrenar su red con una cantidad mucho mayor de datos, sobre los archivos de video directamente, o con otro tipo de redes o combinación de ellas. Se podrían repetir también los análisis mediante algoritmos no supervisados utilizando otro tipo de medidas de similitud, o cambiando la cantidad o la localización de los puntos corporales que se quieren rastrear. Además, sería buena idea realizar el mismo tipo de mediciones y algoritmos con animales realizando diferentes tareas experimentales, para tratar de encontrar patrones de comportamiento que no hemos observado en este trabajo.

Bibliografía

[1] P. Dayan and L. Abbott, *Theoretical Neuroscience*. MIT Press, 2005.

[2] A. C. Müller and S. Guido, *Introduction to Machine Learning with Python*. O'Reilly, 2017.

[3] H. Abdi and L. J. Williams, "Principal component analysis," *WIREs Computational Statistics*, 2010. [Online]. Available: https://doi.org/10.1002/wics.101

[4] B. J. Frey and D. Dueck, "Clustering by passing messages between data points," *Science*, 2007. [Online]. Available: https://doi.org/10.1126/science.1136800

[5] S. J. Prince, *Understanding Deep Learning*. MIT Press, 2023.

[6] E. Stevens, L. Antiga, and T. Viehmann, *Deep Learning with PyTorch*. Manning, 2020.

[7] A. Mathis, P. Mamidanna, K. M. Cury, T. Abe, V. N. Murthy, M. W. Mathis, and M. Bethge, "Deeplabcut: markerless pose estimation of user-defined body parts with deep learning," *Nature Neuroscience*, 2018. [Online]. Available: https://doi.org/10.1038/s41593-018-0209-y

[8] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in python," *Journal of Machine Learning Research*, 2011. [Online]. Available: https://dl.acm.org/doi/10.5555/1953048.2078195

[9] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," *arXiv*, 2017. [Online]. Available: https://doi.org/10.48550/arXiv.1912.01703

[10] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," *IEEE*, 2009. [Online]. Available: https://doi.org/10.1109/CVPR.2009.5206848

[11] A. Klaus, G. J. Martins, V. B. Paixao, P. Zhou, L. Paninski, and R. M. Costa, "The spatiotemporal organization of the striatum encodes action sapce," *Neuron*, 2017. [Online]. Available: https://doi.org/10.1016/j.neuron.2017.08.015

[12] T. Menaker, J. Monteny, L. O. de Beeck, and A. Zamansky, "Clustering for automated exploratory pattern discovery in animal behavioral data," *Frontiers in Veterinary Science*, 2022. [Online]. Available: https://doi.org/10.3389/fvets.2022.884437

Apéndices

A

Apéndice 1

A.1. Salida de la validación cruzada de la red convolucional.

```
Fold 1
-------
Epoch 1, Training loss 0.6801476809713576
Epoch 10, Training loss 0.4564110840200791
Epoch 20, Training loss 0.019115465178627125
Epoch 30, Training loss 0.0004747997609711308
Epoch 40, Training loss 0.00018754709033595048
Epoch 50, Training loss 0.00011584671332081157
Epoch 60, Training loss 8.25112872538001e-05
Epoch 70, Training loss 6.380444031428567e-05
Epoch 80, Training loss 5.1753266570426286e-05
Epoch 90, Training loss 4.3274612362996254e-05
Epoch 100, Training loss 3.701450099254563e-05
Fold 1 Accuracy: 62.96296296296%

Fold 2
-------
Epoch 1, Training loss 0.6727828543495249
Epoch 10, Training loss 0.40820269790128805
Epoch 20, Training loss 0.07222797215251514
Epoch 30, Training loss 0.000436914461994741
Epoch 40, Training loss 0.00018685894079800496
Epoch 50, Training loss 0.00011457800869277382
Epoch 60, Training loss 8.031665121046829e-05
Epoch 70, Training loss 6.14877771731283e-05
```

```
Epoch 80, Training loss 4.955102746876342e-05
Epoch 90, Training loss 4.138665600641302e-05
Epoch 100, Training loss 3.544609912010378e-05
Fold 2 Accuracy: 59.25925925925925%

Fold 3
-------
Epoch 1, Training loss 0.6748640330301391
Epoch 10, Training loss 0.2985653418862192
Epoch 20, Training loss 0.05613466663534702
Epoch 30, Training loss 0.0009909391441616368
Epoch 40, Training loss 0.00021979762501609542
Epoch 50, Training loss 0.00012102077738110442
Epoch 60, Training loss 8.21752691704505e-05
Epoch 70, Training loss 6.146319148839744e-05
Epoch 80, Training loss 4.864212527515648e-05
Epoch 90, Training loss 3.988884605877064e-05
Epoch 100, Training loss 3.376387923664587e-05
Fold 3 Accuracy: 70.37037037037037%

Fold 4
-------
Epoch 1, Training loss 0.6696590812945807
Epoch 10, Training loss 0.2640423259945237
Epoch 20, Training loss 0.4474654574119149
Epoch 30, Training loss 0.0021708315130701075
Epoch 40, Training loss 0.0018092983557858292
Epoch 50, Training loss 0.0016873948513551453
Epoch 60, Training loss 0.0016168329993564929
Epoch 70, Training loss 0.0015647720905907594
Epoch 80, Training loss 0.0015219309327960673
Epoch 90, Training loss 0.0014840632816855294
Epoch 100, Training loss 0.0014494163853304297
Fold 4 Accuracy: 51.85185185185185%

Fold 5
-------
Epoch 1, Training loss 0.6746461634282712
Epoch 10, Training loss 0.2634232545053776
Epoch 20, Training loss 0.0018645723681628217
Epoch 30, Training loss 0.0002821844789475505
Epoch 40, Training loss 0.00014882139868269655
Epoch 50, Training loss 9.860223141253841e-05
Epoch 60, Training loss 7.341558255546943e-05
Epoch 70, Training loss 5.786330583666831e-05
Epoch 80, Training loss 4.7520002915483155e-05
Epoch 90, Training loss 4.0159391091731536e-05
Epoch 100, Training loss 3.450641620334133e-05
Fold 5 Accuracy: 46.913580246913575%

All folds completed!
Mean accuracy: 58.27160493827161%
```