Universidad Autónoma de Yucatán
Facultad de Matemáticas

# Segmentación del parásito de Chagas en imágenes de muestras de sangre mediante ecuaciones diferenciales ordinarias

TESIS

*presentada por:*
*LM Luis Eduardo Ballote Rosado*

*en opción al título de:*
*Maestro en Ciencias de la Computación*

*Directora de tesis:*
*Dra. Anabel Martín González*
*Colaboración: Dr. Carlos Brito Loeza*

*Mérida, Yucatán, México*
*Diciembre 2021*

# Agradecimientos

Quiero agradecer:

A mi asesora, que siempre me brindó consejo cuando hizo falta, y que me proporcionó herramientas para culminar con el trabajo.

Al Dr. Carlos Brito, su constante intervención y ayuda en los temas que abarcó la tesis.

A mi novia, Fernanda, por inspirarme a ser una mejor persona y ser mi mano derecha en la vida; por siempre escucharme cuando lo necesité y sacarme una sonrisa cada que todo se veía gris.

A mis padres, que me han apoyado hasta el cansancio en todas las decisiones de mi vida, y que por más que pasan los años, no dejan de enseñarme lecciones importantes.

A mis amigos, por aguantar mi ausencia en esta laboriosa etapa de mi vida, y por estar para mi en las buenas y en las malas.

II

# Índice general

# Capítulo 1

# Introducción

El mal de Chagas o tripanosomiasis americana es una enfermedad que puede provocar problemas de corazón, megacolon y megaesófago. El causante de esta condición es el parásito *Trypanosoma cruzi* (*T. cruzi*). Pese a que el parásito puede provocar daño severo en su huésped, en la mayoría de los casos, el sujeto infectado puede tomar más de 30 años desde que éste parásito ingresa a su organismo en desarrollar los síntomas [1].

Este padecimiento, es principalmente transmitido por los insectos triatominos, cuyas heces contienen al parásito *T. cruzi*. La tripanosomiasis americana, es la enfermedad endémica más importante en América. Sin embargo, dada la migración de la población de Latinoamérica, se ha esparcido a distintos continentes. Alrededor del 30 % de los infectados desarrollan problemas cardíacos con severas consecuencias.

La enfermedad de Chagas tiene dos fases: la fase aguda y la fase crónica. La primera, ocurre durante las primeras semanas, ésta, por lo general no es detectada debido a que no suelen presentarse síntomas en esta etapa. Sin embargo, incluso en la fase aguda, el mal de Chagas puede ser mortal para niños o para personas con el sistema inmunológico debilitado. La fase crónica por otro lado, le sucede a la fase aguda y en la mayoría de los casos, no se presentan síntomas. No obstante, aproximadamente del 20 % al 30 % de los casos presenta complicaciones cardíacas o gastrointestinales.

## 1.1. Motivación

El diagnóstico de la enfermedad de Chagas, dependerá de la fase en la que se encuentre la infección. Para la fase aguda los métodos parasitológicos son preferidos, y para la fase crónica se opta por los métodos serológicos. Desde las técnicas más simples, como el análisis de frotis sanguíneos bajo el microscopio, pasando por la hemocultura, hasta métodos más sofisticados tales como la reacción en cadena de la polimerasa (PCR), el método más sencillo para el diagnóstico de la enfermedad, es el uso de los frotis de sangre bajo microscopio. Tomando muestras de sangre de 10 $\mu$L, que debe ser presionada para obtener una capa delgada de células de sangre. El análisis de muestras de sangre debe ser cuidadosamente analizado por un experto cuando se siguen estos métodos convencionales, por lo cual, la automatización de esta tarea a través de métodos computacionales puede ayudar a reducir tiempos y costos.

Actualmente, se cuenta con diversas herramientas para detección y/o segmentación de objetos en imágenes, siendo el aprendizaje profundo una opción destacable para desempeñar este trabajo. Por lo que, en este manuscrito, se propone utilizar las ecuaciones diferenciales ordinarias neuronales presentadas en [2] (Neural ODEs). Éstas son una familia de redes profundas, las cuales calculan la salida mediante la resolución de ecuaciones diferenciales y optimizan el gradiente mediante el método de la adjunta [3]. En este trabajo, se hará uso de las Neural ODEs en conjunto con una arquitectura similar a la de la U-Net [4], que es una red especializada en segmentación en imágenes biomédicas, para segmentar al parásito *T. cruzi*. Esto significa, que dada una imagen, se etiquetará cada pixel como parte del parásito, o como parte del fondo. Un ejemplo de segmentación se puede apreciar en la Figura 1.1.

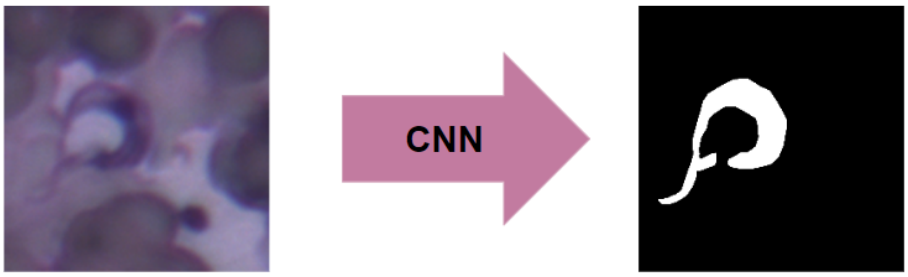



Figura 1.1: Proceso de segmentación mediante una CNN.

## 1.2. Antecedentes

### 1.2.1. Detección y segmentación del parásito del Chagas

La detección y segmentación del parásito de Chagas, es un tema sobre el cuál, se ha escrito una cantidad moderada de artículos. A continuación, se describen algunos de estos.

#### Detección

En 2013, Uc-Cetina *et al.*, realizan una detección del parásito *T. cruzi* en imágenes de muestras de sangre mediante un método que emplea discriminantes gaussianos [5] . El primer paso en esta propuesta consiste en preprocesar las imágenes que pueden o no contener al parásito. Para ello se comienza considerando la imagen original de dimensiones $256 \times 256 \times 3$ pixeles, en formato RGB, y extrayendo de esta última el canal G (verde), obteniendo una imagen en escala de grises de dimensiones $256 \times 256$ pixeles. A continuación, se toma en cuenta que se presenta una acumulación de ácido desoxirribonucleico (ADN) en el parásito y ésta se visualiza como una región de color negro en la imagen. Por lo tanto, se identifican los pixeles con valor mínimo (posibles parásitos), y se analiza su correspondiente subventana de $11 \times 11$ pixeles alrededor de cada punto negro del canal verde. Posteriormente, cada subventana se desdobla en un vector de características de

dimensión 121. En el artículo modelan $p(x|y)$ y $p(y)$, donde $y$ indica si una de las imágenes tiene el parásito $(y = 1)$ o si no lo tiene $(y = 0)$. Luego, se utiliza el teorema de Bayes para calcular

$$p(y|x) = \frac{p(x|y)p(y)}{p(x)}.$$

Los resultados concluyen con 0.0167 falsos negativos, 0.1563 falsos positivos, 0.8437 verdaderos negativos y 0.9833 verdaderos positivos.

En [6] proponen un algoritmo para la detección del *T. cruzi* en imágenes de muestras de sangre. El primer paso es preprocesar las imágenes del conjunto de datos que están en formato RGB. Se realiza una subdivisión de estas imágenes en imágenes de menor tamaño. A continuación, se construye una imagen $BG$ que consiste en las diferencias entre los canales azul y verde, además, se construye una imagen $BGT$ definida de la siguiente manera

$$BGT(x, y) = \begin{cases} 1 & \text{si } |BG(x, y)| > T \\ 0 & \text{de otra manera.} \end{cases}$$

En el artículo se calculó $T = 50$ mediante un método de umbralización local. Posteriormente, se utiliza esta imagen $BGT$ para etiquetar las regiones distintas del fondo en la imagen y se descartan regiones no deseadas mediante discriminación de áreas, debido a la diferencia de tamaños que presentan las células y los parásitos. Para la segmentación, se utiliza un clasificador Gaussiano. Y por último, para la clasificación se emplea un clasificador de k-vecinos más cercanos. La $k$-validación cruzada, concluye una sensitividad de 0.98 y una especificidad de 0.80 para la clasificación.

En [7] se realizó una comparación de dos algoritmos para la detección del parásito *T. cruzi*: el Adaptative Boost (AdaBoost) y Supported Vector Machine (SVM). Estos son algoritmos robustos utilizados para clasificaciones binarias. En este trabajo se introducen características tipo Haar, diseñadas específicamente para la detección del parásito *T.cruzi*. Este tipo de características sirven para brindar información acerca de zonas claras y obscuras en la imagen. El AdaBoost que resultó el mejor método, obtuvo un 100 % en sensitividad y un 93 % en especificidad. La aplicación de AdaBoost con SVM para la detección deseada resultó mejor que cualquiera de los otros dos algoritmos individualmen-

te. El proceso de detección consistió en 4 pasos: (1) adquisición de imágenes mediante el microscopio; (2) conversión de imagen RGB a escala de grises; (3) detección de posibles parásitos utilizando el clasificador AdaBoost previamente entrenado; (4) descartar falsos positivos tomando en cuenta la acumulación de ADN en los parásitos. Este último paso fue implementado con una SVM.

**Segmentación**

En 2014, Soberanis Roger, plantea la segmentación del parásito de Chagas como un problema de clasificación binaria y se proponen algoritmos de clasificación, que aplicados en conjuntos de pixeles conocidos como súper pixeles, resuelven esta tarea [8]. Los clasificadores utilizados fueron SVM, redes neuronales y AdaBoost. En las imágenes que se desea segmentar, después de dividirlas en súper pixeles, se extraen características de cada uno de estos por cada canal de la imagen RGB. En este trabajo se toman en cuenta diversos espacios de color, como el RGB, HSV y YCbCr. Adicionalmente, se consideró un canal formado por la diferencia de los canales azul y verde, como se hizo en [6]. Las características consideradas fueron: aspereza, contraste, direccionalidad, curtosis y varianza. La mayor exactitud se obtuvo considerando como características la curtosis, la media y la varianza en el espacio RGB con el canal de diferencias de azul y verde tomando SVM como clasificador. Bajo estas condiciones se obtuvo una exactitud de 90.75.

En [9] se utiliza una U-Net, que es una red completamente convolucional para la segmentación del *T. cruzi* en imágenes de muestras de sangre. Para realizar esta tarea, se entrena la U-Net con un conjunto de datos que consiste en 974 imágenes a color en formato RGB de tamaño $2560 \times 1920$ pixeles. Las imágenes fueron tomadas de muestras de ratones infectados con el parásito *T. cruzi*. Las imágenes fueron recortadas en 1000 sub-imágenes de tamaño $512 \times 512$ pixeles, Para las cuales se usaron 600 para el conjunto de entrenamiento, 200 para el conjunto de validación y 200 para el conjunto de prueba. Se estudió cómo se comportaba la red con dos funciones de pérdida diferentes: la *Binary Cross-Entropy Loss* (BCE) y una modificación de ésta, la *Weighted Binary Cross-Entropy Loss*(WBCE).

| Modelo | Recall | Precision | F2 score | DC |
|---|---|---|---|---|
| U-Net WBCE | 0.8702 | 0.6304 | 0.8013 | 0.6825 |
| U-Net BCE | 0.7808 | 0.7671 | 0.7714 | 0.7072 |

Tabla 1.1: Resultados obtenidos en [9] (ver Capítulo 3.5.2 donde se discuten las métricas de evaluación).

.

Se realizó aumento de datos para incrementar el número de muestras de entrenamiento. Los resultados obtenidos se pueden observar en la Tabla 1.1.

### 1.2.2. Detección y segmentación del parásito de la Malaria

Debido a la reducida cantidad de artículos acerca de la segmentación del parásito *T. Cruzi*, es pertinente tomar en cuenta las investigaciones afines, como por ejemplo, la segmentación o detección del parásito *Plasmodium*, que es el responsable de la enfermedad de la Malaria, que cuenta un una cantidad numerosa de estudios al respecto.

#### Detección

En 2019, Rahman *et al.*, diseñan diversas redes neuronales convolucionales para la detección del parásito del parásito *Plasmodium* [10]. Primero, se dividen los datos en tres conjuntos, conjunto de entrenamiento, conjunto de validación y conjunto de prueba con una relación de $80:10:10$. Se realiza una $k$-validación cruzada tomando $k = 5$. A continuación, se realizaron diversas técnicas de preprocesamiento en las imágenes del conjunto de datos, las cuales fueron: normalización de la tinción, reescalamiento, estandarización, aumento de datos, de las cuales, los autores destacan que únicamente aumentacón de datos mejora los resultados. Las arquitecturas que se utilizan son denominadas por los autores de la siguiente manera: Custom, TL-VGG16 y CNNEx-SVM. La arquitectura Custom, fue diseñada utilizando 19 capas, con 8 capas convolucionales, 4 capas *maxpool*, 3 capas densas,

| Modelo | Test Accuracy | Precision | F1 score |
|---|---|---|---|
| Custom | 96.29 | 0.9804 | 0.9495 |
| TL-VGG16 | 97.77 | 0.9719 | 0.9709 |
| State-of-the-art Customized | 94.00 | 0.951 | 0.941 |
| State-of-the-art ResNet-50 | 95.70 | 0.969 | 0.957 |
| CNNEx-SVM | 94.77 | 0.9213 | 0.9501 |

Tabla 1.2: Resultados obtenidos en [10].

una capa *flattens*, dos capas con 50 % de *dropout* y una capa completamente conectada. La función de activación usada fue la ReLu, definida como:

$$a = \text{máx}(0, x),$$

donde $a$ es la *output activation* para una entrada $x$ dada. La TL-VGG16 es una modificación de una VGG16 con transferencia de aprendizaje, usando pesos preentrenados en el ImageNet [11]. Finalmente, en la CNNEx-SVM, se utiliza una red neuronal convolucional (CNN) para extraer características y se emplea una SVM como clasificador usando las características profundas extraídas. En el artículo, también se considera el *método combinado*, que consiste en utilizar las tres redes independientemente y tomar el promedio de las predicciones obtenidas de estas redes. Los resultados se muestran en la Tabla 1.2.

**Segmentación**

En [12] y en [13] se utiliza una U-Net para segmentar al parásito de la malaria. En particular, en [12], se usa un conjunto de datos de 30 imágenes RGB y se recurre a técnicas de aumento de datos para formar un conjunto de datos de 500 imágenes. Éstas son normalizadas para el entrenamiento de la U-Net. Además, en [12] se comparan las funciones de pérdida, error cuadrático medio (MSE), entropía binaria cruzada (BCE) y la función de pérdida de Huber. Los mejores resultados se obtuvieron para la función de

| Clase | Dice Score | Precision | Recall | F1 score |
|---|---|---|---|---|
| Promastigote | 0.495 | 0.512 | 0.476 | 0.491 |
| Adhered | 0.707 | 0.677 | 0.379 | 0.457 |
| Amastigote | 0.777 | 0.757 | 0.823 | 0.777 |

Tabla 1.3: Resultados obtenidos en [14].

.

pérdida de Huber con una F1-score de 0.9297 y una sensitividad de 0.8957. Por otro lado, en [13] se obtuvo 0.98 de precisión.

### 1.2.3. Segmentación del parásito de la Leishmaniasis

En [14] se presenta una U-Net para la segmentación de parásitos de Leishmania, para después clasificarlos en tres tipos de parásitos. La red combina una red convolucional con una red deconvolucional. La red consiste en aplicar convoluciones con filtros de $3 \times 3$ seguidas de la función de activación ReLU y luego un maxpooling con tamaño de paso 2. Ésta arquitectura tiene 23 capas convolucionales y la última capa es usada para obtener la segmentación. El entrenamiento consiste de 37 imágenes, cada una con su correspondiente verdad fundamental con siete etiquetas. El conjunto de datos tiene gran desbalance de clases, puesto que la mayoría de los pixeles corresponden al fondo. Para solucionar el problema de desbalance se utiliza como función de costo la *Generalized Dice Loss*. Los resultados de la segmentación se pueden observar en la Tabla 1.3.

Para encontrar y optimizar el gradiente se utiliza el método de la adjunta. Entre las ventajas de este método se encuentran las siguientes: escala linealmente con el tamaño del problema, tiene bajo costo de memoria y controla explícitamente el error numérico.

## 1.3. Objetivos

### 1.3.1. Objetivo general

El objetivo general de esta tesis es implementar y analizar el desempeño de una ecuación diferencial ordinaria para segmentar al parásito *T. cruzi* en imágenes de muestras de sangre.

### 1.3.2. Objetivos específicos

1. Implementar un modelo computacional basado en Neural ODEs para segmentar al parásito *T. cruzi*.

2. Evaluar los resultados y compararlos con métodos del estado del arte.

3. Redactar la tesis y publicar los resultados.

### 1.3.3. Organización de la tesis

En el Capítulo 1 se da una introducción al problema que se quiere resolver y se exploran las motivaciones y los trabajos previos relevantes para ésta problemática.

Luego, en el Capítulo 3 se abordan los temas más importantes que constituyen las bases para los capítulos posteriores, comenzando por la Sección 3.1 con una introducción al aprendizaje de máquina, qué es, sus principales paradigmas y las notaciones generales. Se hará énfasis el área del aprendizaje supervizado en 3.2 y se avanzará utilizando un ejemplo: una regresión lineal, en la cual, se seguirán explorando conteptos fundamentales del aprendizaje de máquina, como son: conjuntos de entrenamiento y de prueba, optimización, descenso de gradiente, sobreajuste y regularización. Posteriormente, en la Sección 3.4, se presentará la notación que se utilizará a lo largo del escrito. Por último, en este capítulo se introducirá el problema de la segmentación semántica en la Sección 3.5.2, junto con las métricas apropiadas para evaluar éste problema.

En el Capítulo 4 se presentan los conceptos relacionados con el aprendizaje profundo, como las convoluciones, redes neuronales convolucionales, el entrenamiento de éstas etc. También se exploran los paradigmas principales que se utilizan en este trabajo en las Secciones 4.3 y 4.4.

Posteriormente, en el Capítulo 5 se abarca todo lo relacionado con las Ecuaciones Diferenciales Ordinarias Neuronales.

Por último, en el Capítulo 6 se muestran los experimentos realizados y los resultados obtenidos.

# Capítulo 2

# Metodología

Para realizar la segmentación del *T. cruzi*, en este trabajo se utilizará una Neural ODE basada en la arquitectura U-Net. Particularmente se tomará como base el modelo propuesto en [15]. A continuación, se describe la arquitectura y sus elementos. Antes de explicar las Neural ODEs, se hará una breve revisión de las ResNet como motivación natural para definirlas.

## 2.1. ResNet

Las ResNet son un tipo de redes neuronales convolucionales profundas cuya característica definitoria es que presenta conexiones de salto cuya representación gráfica se puede observar en la Figura 2.1.

Es decir, una ResNet es una red compuesta por bloques, que normalmente son pequeñas redes convolucionales (e.g. 2 convoluciones), tales que la entrada del siguiente bloque es la suma de la salida y la entrada del anterior. O matemáticamente, si $f_{t-1}$ es el t-ésimo bloque, y $\mathbf{z}_t$ es la entrada del t-ésimo bloque, entonces se tiene que

$$\mathbf{z}_t = \mathbf{z}_{t-1} + f_{t-1}(\mathbf{z}_{t-1}; \theta_t). \tag{2.1}$$

En el caso particular en el que todos los $f_t$ son iguales, se simplifica ligeramente a la

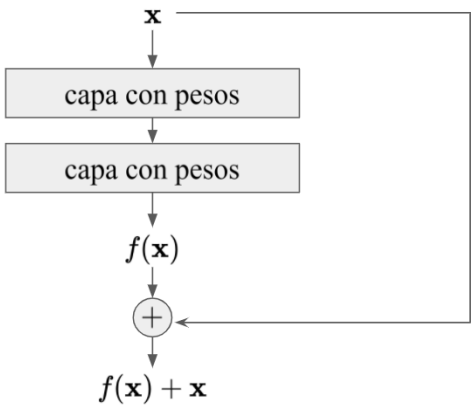



Figura 2.1: Bloques residuales [16].

ecuación

$$\mathbf{z}_t = \mathbf{z}_{t-1} + f(\mathbf{z}_{t-1}; \theta_t).$$

## 2.2. Ecuaciones Diferenciales Ordinarias Neuronales

Las **Ecuaciones Diferenciales Ordinarias Neuronales** (Neural ODEs), fueron propuestas en 2018 por Chen *et al.*; ganaron ese mismo año el título al mejor artículo en la *Conference on Neural Information Processing Systems* (NeurIPS). Las Neural ODEs son una nueva familia de redes neuronales basadas en ecuaciones diferenciales ordinarias.

Considérese una ResNet $R$ con $T$ bloques residuales, donde la salida de cada uno de ellos está dada por la ecuación (2.2). Se observa que esta ecuación es similar al método de Euler para resolver ecuaciones diferenciales. Si se añaden más bloques y se toman pasos más pequeños, se puede aproximar en el límite la siguiente ecuación:

$$\mathbf{z}(t) = \frac{d}{dt} f(\mathbf{z}(t), t; \theta), \quad \mathbf{z}(0) = \mathbf{x}, \quad t \in [0, T]. \tag{2.2}$$

Se puede usar cualquier algoritmo de ecuaciones diferenciales para encontrar $\mathbf{z}(T)$, que es la salida de $R$.

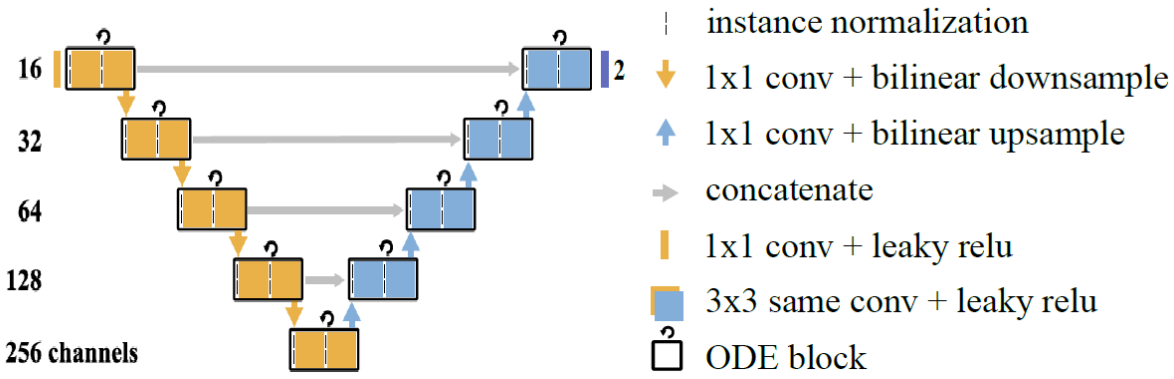



Figura 2.2: Arquitectura UNODE. Imagen extraída de [15].

## 2.3. UNODE

La UNODE [15], así como su predecesora, la U-Net, consta de un camino contractivo y de uno expansivo donde cada camino consiste en sus respectivos bloques. La primera capa es una convolución $1 \times 1$ con una función de activación leaky ReLu, definida por

$$f(x) = \begin{cases} x & \text{si } x > 0 \\ 0.01x & \text{si } x \leq 0. \end{cases}$$

Después, empieza el camino contractivo, que consta de cinco bloques que están formados por una normalización de instancias, seguido de una NODE (bloque ODE) y un submuestreo bilineal. Continua con el camino expansivo, donde cada uno de los cuatro bloques respectivos, consiste en una normalización de instancias seguido de un bloque ODE y un sobremuestreo bilineal. Se puede apreciar ésta arquitectura en la Figura 2.2.

# Capítulo 3

# Marco Teórico

## 3.1. Aprendizaje de Máquina

El **aprendizaje de máquina** (**ML**) en palabras generales es un área de la computación que consiste en resolver tareas dadas mediante algoritmos que hagan uso de la experiencia. En palabras de Tom Mitchell (traducidas del inglés) en 1997, tenemos la siguiente definición:

> Una computadora dice que aprende de la experiencia $E$ con respecto a una tarea $T$ y una medida de desempeño $P$, si su desempeño en $T$ medido por $P$ mejora con la experiencia $E$.

El aprender de la experiencia puede tener varias connotaciones, por ejemplo un algoritmo que requiera clasificar gatos y perros podrá requerir haber "visto" miles o millones de fotos etiquetadas con su respectiva clase (perro o gato) para con ello "aprender" a distinguir qué características corresponden a un perro y qué características corresponden a un gato. Un algoritmo que aprenda a jugar ajedrez, requerirá jugar muchas partidas y qué movimientos resultaron ganadores o perdedores y "decidir" su estrategia de juego con base en eso.

Algunas ejemplos adicionales de aplicaciones del aprendizaje de máquina son: detectar tumores en escaneos del cerebro, detectar comentarios inapropiados en redes sociales, crear

un asistente personal digital, detectar fraudes de pagos no reconocidos de una tarjeta de crédito, predecir el clima basado en ciertas características, etc.

Hay tres paradigmas principales en el área del aprendizaje automático: aprendizaje supervisado, aprendizaje no supervisado y aprendizaje por refuerzo. El **aprendizaje supervisado** consiste en entrenar un algoritmo con muestras previamente etiquetadas; nuestro problema de interés cae en éste paradigma por lo cuál explicaremos más a detalle el marco de trabajo de éste. El **aprendizaje no supervisado** está conformado por algoritmos donde los datos de entrenamiento no están previamente etiquetados. Muchos de estos algoritmos tienen como objetivo agrupar datos en grupos similares o clusters. Por último, en el **aprendizaje por refuerzo**, conseguiremos nuestra meta otorgándole recompensas positivas a un agente si realiza correctamente el trabajo y recompensas negativas si por el contrario realiza incorrectamente su tarea.

## 3.2. Aprendizaje Supervisado

Tenemos un problema de aprendizaje supervisado cuando se quiere aprender de datos con etiquetas, es decir, nos dan un conjunto de parejas $A = \{(x^{(m)}, y^{(m)})\}_{m=1}^{M}$, con $x^{(m)} \in \mathcal{X}$, $y^{(m)} \in \mathcal{Y}$. A los conjuntos $\mathcal{X}$ y $\mathcal{Y}$ los llamaremos **espacio de entrada** y **espacio de salida** respectivamente, también a las $x^{(m)}$ les llamaremos **las entradas** y a las $y^{(m)}$ les llamaremos **los objetivos**. Suponemos que existe una función $f : \mathcal{X} \to \mathcal{Y}$ tal que $f(x^{(m)}) = y^{(m)}$ y nuestro objetivo es encontrar una estimación $\widehat{f} \in \mathcal{H}$ de $f$, *i.e.* $\widehat{f}(x) \approx f(x)$ para toda $x \in \mathcal{X}$, en particular $\widehat{f}(x^{(n)}) = y^{(n)}$. Al conjunto $\mathcal{H}$ se le conoce como el **espacio de hipótesis** y en la práctica suele estar parametrizado por $\boldsymbol{\theta} \in \mathbb{R}^{N_\theta}$, de manera que $\mathcal{H}$ estará constituida exactamente por funciones $f(\cdot; \boldsymbol{\theta})$. Nos encontraremos con la notación $\widehat{y}$ para denotar la estimación de una salida $y$, por ejemplo, si contamos con el estimador $\widehat{f}$, tendríamos $\widehat{y}^{(m)} = f(x^{(m)})$.

Para conseguir nuestro objetivo de hallar la mejor estimación de $\widehat{f}$, los problemas de aprendizaje supervisado asocian al problema una función de pérdida $L : \mathbb{R}^{N_y} \times \mathbb{R}^{N_y} \to \mathbb{R}$. Esta función, representa el error entre la predicción $\widehat{y} = \widehat{f}(x)$ con $y = f(x)$. Nuestro

problema se traduce entonces en reducir el error en todo $A$, es decir queremos hallar

$$\arg\min_{\widehat{f}\in\mathcal{H}} \sum_{m=1}^{M} L\left(\widehat{f}\left(x^{(m)}\right), y^{(m)}\right). \tag{3.1}$$

En un modelo paramétrico $f(\cdot, \boldsymbol{\theta})$, tendremos que $L(\widehat{y})$ es una función de $\boldsymbol{\theta}$ ya que $\widehat{y} = f(x; \boldsymbol{\theta})$, por lo que es conveniente definir $J : \mathbb{R}^{N_\theta}$ como

$$J(\boldsymbol{\theta}) = \frac{1}{M} \sum_{m=1}^{M} L\left(\widehat{f}\left(x^{(m)}\right), y^{(m)}\right). \tag{3.2}$$

Nuevamente, podemos escribir el problema como un problema de optimización

$$\arg\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta}). \tag{3.3}$$

La minimización y maximización de funciones es un área de las matemáticas ampliamente estudiada que lleva el nombre de **optimización**. Podemos tomar los métodos que nos ofrece esta área, para realizar el ajuste de nuestros parámetros y encontrar el mejor modelo (en el caso paramétrico).

En ocasiones estaremos interesados en un subconjunto $B = \{(x^{(i)}, y^{(i)})\}_{i\in I} \subset A$, donde $I \subset \{1, ..., M\}$, definimos **el error en** $B$ como

$$L_B = \frac{1}{|I|} \sum_{i\in I} L\left(\widehat{y}^{(i)}, y^{(i)}\right),$$

y también, cuando $\widehat{f}$ es paramétrica $J_B(\boldsymbol{\theta}) = L_B$.

En diversas ocasiones, nos encontraremos con problemas donde nuestro objetivo sea maximizar o minimizar alguna métrica $P$, sin embargo nosotros minimizaremos $L$ con la esperanza de que hacer esto minimice o maximice $P$ según requiramos. Cuando alguna métrica $P$, también denotaremos $P_B$ como el promedio de $P$ en el conjunto $B$.

Para finalizar con las notaciones generales, en ocasiones nuestras entradas, serán vectores o imágenes que son representadas por matrices o por tensores. En cualquiera de estos casos escribiremos la variable en negritas: $\mathbf{x}$, haremos esto también si pasa con los objetivos *i.e.* escribiremos $\mathbf{y}$. En ocasiones nos interesaremos en el tensor $\mathbf{X} = (\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(M)})$ o

bien si las entradas están en $\mathbb{R}$, denotamos $\mathbf{x} = (x^{(1)}, x^{(2)}, ..., x^{(M)})$. En los casos más comunes, las entradas son multidimensionales y las salidas unidimensionales, de manera que tendríamos al conjunto $A$ denotado como $\mathbf{X}$ y $\mathbf{y}$. En el problema de la segmentación del *T. Cruzi* nuestros objetivos son imágenes de 1 canal, así que el conjunto $A$ estaría denotado como $\mathbf{X} \in \mathbb{R}^{M\times 3\times H\times W}$ y $\mathbf{Y} \in \mathbb{R}^{M\times 1\times H\times W}$. Nuevamente, en ocasiones estamos interesados en un subconjunto de $B$, y definiríamos

$$\mathbf{X}^{(B)} = (\mathbf{x}^{(i_1)}, ..., \mathbf{x}^{(i_K)}),$$

donde $I = \{i_1, ..., i_K\}$.

## 3.3. Regresión Lineal

En el aprendizaje supervisado encontramos dos problemas principales: la **regresión** y la **clasificación**. La regresión tiene lugar cuando queremos predecir variables continuas y la clasificación cuando queremos predecir variables categóricas, es decir, variables discretas no ordenadas. También podemos predecir variables discretas ordenadas o variables **ordinales**, dependiendo del contexto esto entrará dentro de la categoría de regresión o de clasificación. El problema de interés de este trabajo no entra en ninguna de estas categorías *per se*, si no más bien es una tarea de segmentación que definiremos más adelante. Pese a esto, abordaremos un problema de regresión, pues es suficientemente sencilla para exponer los conceptos principales que hemos presentado y definir algunos más que se generalizan o adaptarán fácilmente en nuestro problema y nos familiarizamos con el marco del aprendizaje de máquina.

Empecemos con una regresión sencilla, donde $\mathcal{X}, \mathcal{Y} \subset \mathbb{R}$. Podemos graficar los puntos $(x^{(m)}, y^{(m)})$ en el plano como se observa en la Figura 3.1.

El ejemplo con el que vamos a trabajar fue generado artificialmente, pero simularemos que el eje $X$ representa alturas en pulgadas y el eje $Y$ representa pesos en libras. Contamos con 20 muestras que suponemos que corresponden a mediciones de estas cantidades de 20 personas distintas. El primer paso es entender el problema que queremos resolver. En este

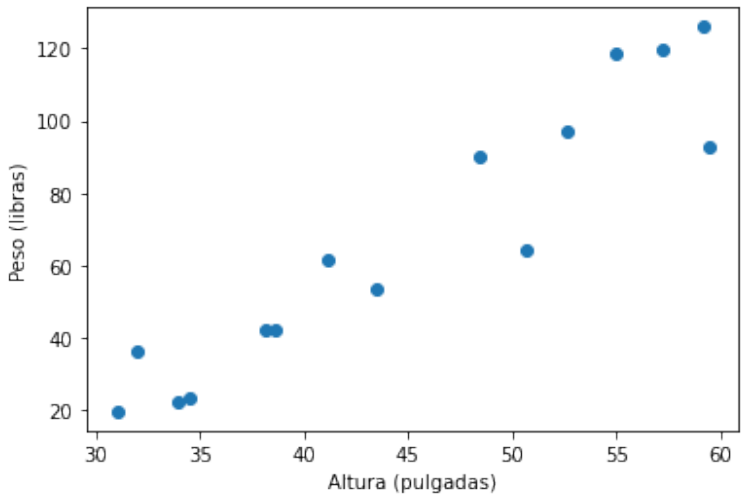



Figura 3.1: Ejemplo de aprendizaje supervisado, queremos predecir el peso, dada la altura.

caso queremos generar un modelo que haga predicciones, de manera que si nos dan una altura, podamos estimar el peso. Estamos suponiendo que el peso comparte una relación con la altura, tal vez, con cierto grado de error, es decir que existe una función $f : \mathcal{X} \to \mathcal{Y}$, donde $\mathcal{X} \subset \mathbb{R}$ es el conjunto de todas las posibles alturas y $\mathcal{Y}$ es el conjunto de todos los posibles pesos (o por simplicidad $\mathcal{X} = \mathcal{Y} = \mathbb{R}$), y también estamos suponiendo dada una altura $x$, su peso va a ser $f(x) + \epsilon$, donde $\epsilon$ representa el grado de error.

Ahora que conocemos el problema, procedemos a definir un modelo para realizar las predicciones. En este caso, si vemos la Figura 3.1 vemos que los puntos parecen estar dispuestos en una línea recta en el plano con cierto grado de error. Tomando esto en cuenta, parece apropiado considerar un modelo de la forma

$$\widehat{f}(x; w, b) = wx + b$$

Este modelo es un ejemplo de un **modelo lineal**. En este caso nuestros parámetros son $w, b \in \mathbb{R}$ (*i.e.* $\theta = (w, b)^\intercal$). Nuestro objetivo, como ya mencionamos es ajustar estos parámetros de manera que minimicemos una función de pérdida $L$. La función de pérdida $L$ debe ser definida de tal manera que minimizar $L$ con respecto a los parámetros, también resuelva el problema. En nuestro caso, queremos encontrar un número real $y$, así que

nuestra primera idea de función de pérdida puede ser simplemente

$$L(\widehat{y}, y) = |y - \widehat{y}|$$

de manera que al minimizar $L$, $\widehat{y}$ se acerque lo más posible al objetivo $y$. A esta función de pérdida se le llama **error absoluto**, y cuando consideramos el promedio en todo el conjunto $A = \{(x^{(m)}, y^{(m)})\}_{m\epsilon\mathcal{M}}$

$$\text{MAE} = \frac{1}{M} \sum_{m\epsilon\mathcal{M}} \left|y^{(m)} - \widehat{y}^{(m)}\right|$$

se le conoce como **error absoluto medio** (**MAE**). Otra función de pérdida común, que penaliza más fuertemente los errores grandes es el **error cuadrado**:

$$L(\widehat{y}, y) = (y - \widehat{y})^2,$$

la cual por supuesto da lugar a el **error cuadrado medio** (**MSE**):

$$\text{MSE} = \frac{1}{M} \sum_{m\epsilon\mathcal{M}} \left(y^{(m)} - \widehat{y}^{(m)}\right)^2.$$

Por el momento, seleccionaremos el error cuadrado como nuestra función de pérdida para el ejemplo. El siguiente paso es justamente minimizar el MSE, ajustando nuestros parámetros $w, b$.

Como ya mencionamos, minimizar $\arg\min J(w, b)$ es un problema de optimización, del cual tenemos una amplia diversidad de métodos a nuestra disposición. En un caso tan simple como esta regresión lineal, se pueden encontrar los parámetros óptimos $w^*$ y $b^*$ analíticamente, sin embargo, con el fin de introducir las ideas principales de los métodos de optimización actuales, en su lugar, resolveremos este problema utilizando el método del **descenso de gradiente**.

### 3.3.1. Descenso de gradiente

El descenso de gradiente es un método iterativo de optimización cuya idea principal es en cada paso tomar la dirección donde la función a optimizar desciende lo más rápido

posible. Esto ocurre justamente en la dirección opuesta al gradiente, de manera que si queremos minimizar $J(\boldsymbol{\theta})$, la condición de actualización en cada paso quedaría

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\theta} J(\boldsymbol{\theta}),$$

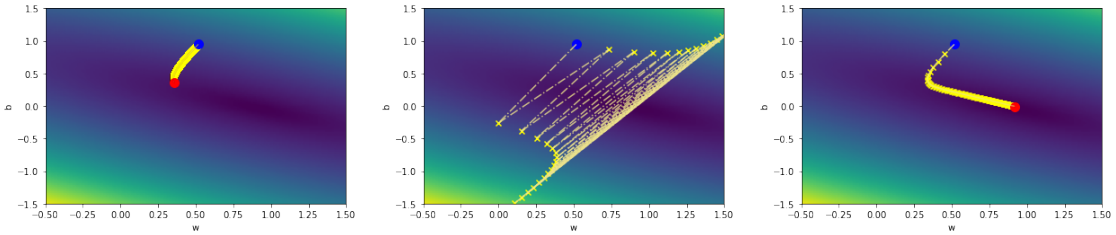
aquí $\alpha$ es una constante en $\mathbb{R}$ y se le conoce como **razón de aprendizaje**. Esta constante regula la rapidez a la que nuestro algoritmo convergerá: si escogemos $\alpha$ muy pequeña relativa al problema, entonces convergerá en muchos pasos, mientras que si escogemos $\alpha$ muy grande, el algoritmo puede ni siquiera converger. Así que es importante escoger adecuadamente esta $\alpha$ en un punto justo donde podamos garantizar la convergencia en la menor cantidad de pasos posibles (bajo ciertas condiciones de $f$ que en la práctica suelen cumplirse y con un $\alpha$ suficientemente pequeña, se puede garantizar la convergencia del método [17]). Por otra parte, cabe mencionar que aunque garanticemos la convergencia, ésta únicamente es local, lo que significa que no siempre hallaremos los parámetros óptimos, esto en la práctica puede representar un problema y por lo tanto se han diseñado otros algoritmos basados en el descenso del gradiente que son suficientemente robustos como para evitar estos óptimos locales (*e.g.* ADAM [18]). Por motivos de simplificación, antes de aplicar el algoritmo, escalaremos los datos al intervalo $[0, 1]$. En particular utilizando el siguiente método:

$$x_{\text{scaled}} = \frac{\text{mín}_m \, x^{(m)}}{\text{máx}_m \, x^{(m)} - \text{mín}_m \, x^{(m)}}$$

$$y_{\text{scaled}} = \frac{\text{mín}_m \, y^{(m)}}{\text{máx}_m \, y^{(m)} - \text{mín}_m \, y^{(m)}}.$$

A este tipo de transformación, se le conoce como **normalización** y existen varios métodos para realizarlo y se suele usar para que todos los datos de entrada estén en la misma escala. No es convencional normalizar la variable objetivo, y en este caso, como sólo contamos con una variable de entrada. Tampoco es útil para fines del entrenamiento escalar $\{x^{(m)}\}_{m \in \mathcal{M}}$, y sólo se hará para mayor claridad en la visualización del descenso del gradiente.

Volviendo al ejemplo, mencionamos que para el descenso del gradiente, tenemos que escoger una razón de aprendizaje $\alpha$, adecuado. En la Figura 3.2 se puede apreciar la



(a) $\alpha = 0.0001$. (b) $\alpha = 0.8$. (c) $\alpha = 0.1$.

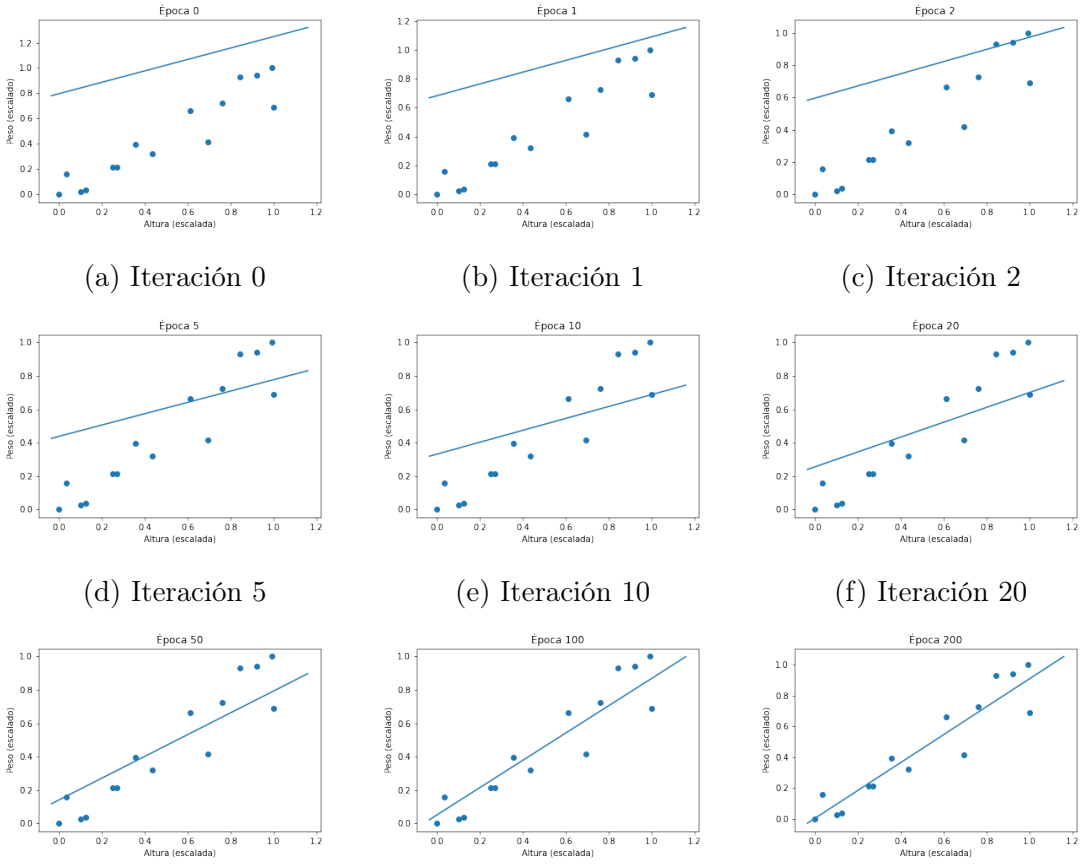
Figura 3.2: En todos los casos $w$ y $b$ tienen el mismo valor inicial (punto azul) y se corren 1000 épocas (iteraciones del descenso de gradiente). El punto rojo es el último al que se llegó (a) Cuándo $\alpha$ es muy pequeño se acercará lentamente al punto óptimo. (b) Cuando $\alpha$ es muy grande puede diverger. (c) Aquí vemos una buena selección de $\alpha$, donde llega rápidamente al punto óptimo.

función de pérdida del ejemplo graficada y los pasos que hizo el descenso de gradiente con distintas razones de aprendizaje.

En la Figura 3.3 se puede ver cómo se va ajustando la línea conforme avanzan las iteraciones del descenso de gradiente.

Con esto *a priori* habríamos terminado de resolver el problema del ejemplo pues después de aplicar el descenso de gradiente, ya habríamos encontrado $w^*$ y $b^*$. En nuestro ejemplo particular, tenemos una función convexa, lo cual garantiza la existencia de mínimo global. Antes de continuar con la discusión del ejemplo, justificaremos la intuición detrás del descenso de gradiente.

Supongamos que queremos minimizar una función $f(\mathbf{x})$. La idea del descenso de gradiente es tomar la dirección $\mathbf{d}$ para la cuál $f(\mathbf{x}+\epsilon\mathbf{d})$ es mínimo para $\epsilon > 0$ muy pequeño y $\mathbf{d}$ es un vector unitario. Esto corresponde con la derivada direccional de $f$ en la dirección




(a) Iteración 0 (b) Iteración 1 (c) Iteración 2

(d) Iteración 5 (e) Iteración 10 (f) Iteración 20

(g) Iteración 50 (h) Iteración 100 (i) Iteración 200

Figura 3.3: El modelo utilizando los parámetros ajustados después de distintos números de iteraciones.

$\mathbf{d}$ la cual es igual a $\mathbf{d}^\top \nabla_{\mathbf{x}} f(\mathbf{x})$, de esta forma queremos encontrar la dirección

$$\begin{aligned}\mathbf{d}^* &= \arg\min_{\mathbf{d},\|\mathbf{d}\|=1} \mathbf{d}^\top \nabla_{\mathbf{x}} f(\mathbf{x}) \\ &= \arg\min_{\mathbf{d},\|\mathbf{d}\|=1} \|\mathbf{d}\| \ \|\nabla_{\mathbf{x}} f(\mathbf{x})\| \cos\phi \\ &= \arg\min_{\mathbf{d},\|\mathbf{d}\|=1} \cos\phi,\end{aligned}$$

donde $\phi$ es el ángulo formado entre $\mathbf{d}$ y $\nabla_{\mathbf{x}} f(\mathbf{x})$. Se minimiza la expresión $\cos\phi$ cuando $\phi = \pi$, es decir, $\mathbf{d}$ y $\nabla_{\mathbf{x}} f(\mathbf{x})$ tienen direcciones opuestas: $\mathbf{d} = -\nabla_{\mathbf{x}} f(\mathbf{x})$.

### 3.3.2. Sobreajuste

Como mencionamos, resolvimos satisfactoriamente el problema de optimización que planteamos, sin embargo, en la práctica, hallar el punto óptimo no es nuestro problema principal, sino más bien, un problema secundario que esperamos que al resolverlo, también hallemos respuesta a nuestras verdaderas preguntas. Por ejemplo, en este caso podríamos querer predecir el peso de una persona con respecto a la altura. Como mencionamos antes queremos hallar una función $\widehat{f}$ que se aproxime a una función $f$ que suponemos que existe en todo $\mathcal{X}$, no únicamente en una muestra.

Un procedimiento que ilustra el inconveniente que surge de optimizar los parámetros en toda la muestra es el siguiente: podemos cambiar el modelo a uno más complejo, por ejemplo, uno de la forma

$$\widehat{f}(x) = w_{11}x^{11} + w_{10}x^{10} + \ldots + w_1 x + b.$$

Éste puede ser ajustado con un MSE todavía menor al del modelo lineal (Figura 3.4), o aún peor, un polinomio de grado 15 puede ajustarse de tal manera que $\text{MSE}(\widehat{y}, y)$ sea igual a 0, sin embargo, la tendencia de este conjunto de datos es claramente lineal.

Claro que esta función sería la que estamos buscando si nuestro problema fuera simplemente optimizar los parámetros, sin embargo el problema real es estimar una función $f$ que describe la tendencia de toda la población, no sólo de una muestra. Al fenómeno que

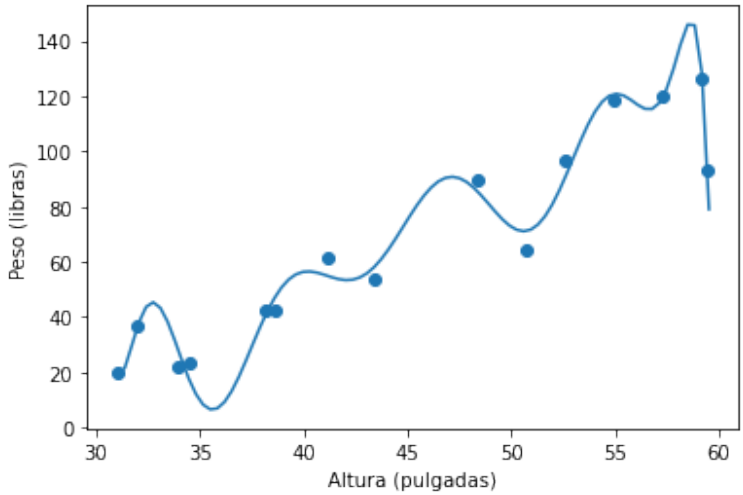



Figura 3.4: Un modelo polinomial sobreajustando los datos.

presenta este modelo, se le conoce como **sobreajuste** y existen estrategias para detectarlo (lo cual es necesario cuando contamos con más dimensiones) y para evitarlo.

Para elegir mejor nuestro modelo y evitar el sobreajuste, una de las opciones es aumentar el número de muestras, volviendo al ejemplo, si contáramos con 20 datos, en lugar de 15, veríamos que el polinomio que obtuvimos antes daría una pérdida promedio mayor incluso que a la del modelo lineal. De hecho, podemos apreciar lo que acabamos de discutir en la Figura 3.5.

No siempre es posible adquirir más muestras, sin embargo podemos simular el proceso anterior, particionando el conjunto $A$, en dos conjuntos a los que llamaremos **conjunto de entrenamiento**, que utilizaremos para ajustar los parámetros, y **conjunto de prueba** en el cual evaluaremos qué tan bien se desempeña el modelo en puntos que nunca ha "visto". A estos conjuntos los denotaremos $A_{\text{train}} = (\mathbf{X}^{(\text{train})}, \mathbf{y}^{(\text{train})})$ y $A_{\text{test}} = (\mathbf{X}^{(\text{test})}, \mathbf{y}^{(\text{test})})$. Supondremos que originalmente teníamos menos puntos (en el ejemplo 12), y que luego obtuvimos nuevos puntos (en este caso 3). En la Figura 3.6, podemos ver el conjunto de datos separados y cómo nuevamente no generaliza bien para datos nuevos.

**Nota 3.1.** *Este es un ejemplo de juguete y probablemente otras técnicas como el K-Fold validation, funcionarían mejor para evaluar el desempeño del modelo. El objetivo de esta*

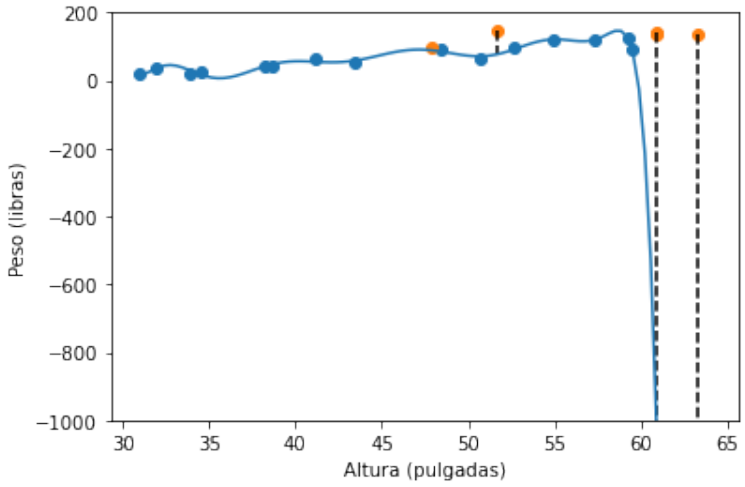



Figura 3.5: Proyecciones nuevos datos a la curva sobreajustada.

*sección fue meramente introducir el concepto de sobreajuste.*

El particionar el conjunto como hemos visto, es útil para detectar cuando el modelo está haciendo un sobreajuste y poder tomar medidas al respecto. Entre las medidas que podemos tomar están por ejemplo: tomar más muestras, reducir el número de características, escoger un modelo más simple o agregar un regularizador a la función de pérdida. Este último método lo explicaremos más a detalle a continuación. Continuando con el ejemplo, supongamos que escogimos el modelo polinomial para modelar nuestro problema, aunque no es el mejor modelo que podemos utilizar, nos servirá para ejemplificar como usar un regularizador. Entre los problemas de aprendizaje supervisado se encuentra nuestro problema a resolver: **segmentación semántica** o simplemente segmentación para fines de este trabajo. En este capítulo nos dedicaremos a la discusión de este problema, así como las métricas de evaluación de resultados y posibles funciones de pérdida para esta tarea [19, 20].

La segmentación de un objeto en una imagen, consiste en encontrar los píxeles correspondientes del objeto en la imagen. En la figura 3.7, podemos observar un ejemplo de segmentación de una imagen. Esta tarea puede ser generalizada a la segmentación de varios objetos en la imagen; un ejemplo se puede ver en la Figura 3.8.

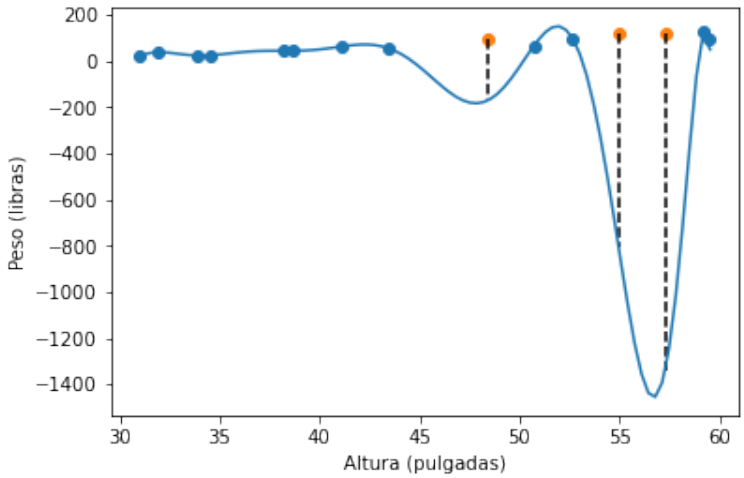



Figura 3.6: Proyecciones del conjunto de prueba a la curva sobreajustada.

Figura 3.7: Imagen original, el objeto a segmentar es el avión (izquierda). Verdad Fundamental, los píxeles del avión están coloreados de rojo y los píxeles del fondo de negro (derecha). Imagen extraída de [21].

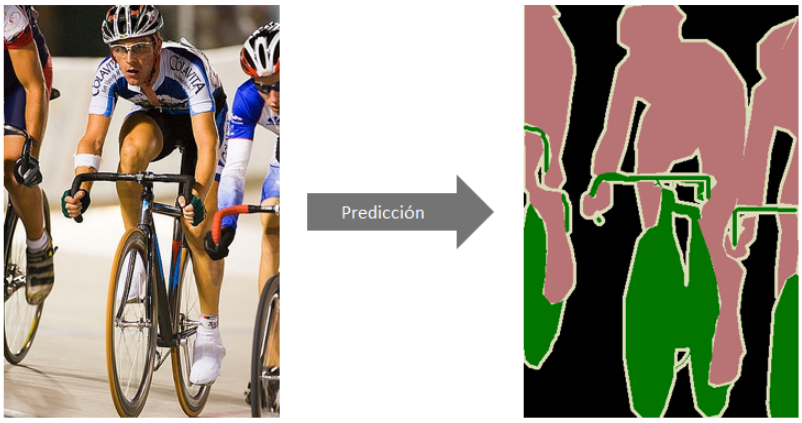



Figura 3.8: Imagen original, vemos dos objetos, la persona y la bicicleta (izquierda). Los píxeles de la persona coloreados de rosa, los de la bicicleta de verde, y el fondo de negro (derecha). Imagen extraída de [22].

## 3.4. Notaciones generales

Antes de continuar con la descripción matemática de la segmentación, introduciremos notaciones generales que usaremos a lo largo del trabajo. Debido a la naturaleza del problema, estaremos trabajando con imágenes, o más generalmente con **tensores** que son elementos en $\mathbb{R}^{C\times H\times W}$, donde $C$ son los **canales**, $H$ es la **altura** del tensor o el número de filas y $W$ es el **ancho** del tensor o el número de columnas. En algunos contextos nos interesaremos únicamente por matrices en $\mathbb{R}^{H\times W}$ que identificaremos con $\mathbb{R}^{1\times H\times W}$; las imágenes en escala de grises entran dentro de esta categoría. Las imágenes a color en formato RGB son un caso particular de tensor donde $C = 3$ y de hecho cada entrada puede tomar el valor de un entero entre 0 y 255 inclusive. Por último, en el contexto del aprendizaje automático, algunos algoritmos requieren de un lote de $B$ tensores, lo cual forma un tensor en $\mathbb{R}^{B\times C\times H\times W}$.

## 3.5. Segmentación

Si tenemos una imagen $\mathbf{I} \in \mathbb{R}^{C\times H\times W}$ y únicamente un objeto y el fondo (es decir estamos ante una segmentación binaria). La segmentación consiste entonces, en encontrar los píxeles del objeto $O \subset \{1,2,...,H\}\times\{1,2,...,W\}$, y definir la nueva imagen segmentada $\mathbf{I}_O \in \{0,1\}^{H\times W}$ de la siguiente manera:

$$\mathbf{I}_{Oi,j} = \begin{cases} 1 & \text{si } (i,j) \in O \\ 0 & \text{si } (i,j) \notin O. \end{cases} \tag{3.4}$$

En la práctica no siempre es posible encontrar los píxeles exactos, por lo que encontraremos una estimación $\widehat{O}$ y definiremos de manera análoga a (3.4) una imagen $\mathbf{I}_{\widehat{O}}$. La imagen $\mathbf{I}_O$ es la ***verdad fundamental***, es decir, la imagen que nos gustaría obtener, y la imagen $\mathbf{I}_{\widehat{O}}$ es nuestra **predicción**. A partir de ahora, por simplicidad y siguiendo las convenciones en el área del aprendizaje automático, siempre que quede claro el contexto, denotaremos $\mathbf{y}$ a $\mathbf{I}_O$, y así mismo, $\widehat{\mathbf{y}}$ a $\mathbf{I}_{\widehat{O}}$.

### 3.5.1. Métricas para la segmentación

Comentamos en la Sección 3.1 que normalmente nos interesa optimizar métricas distintas a la función de pérdida que optimizamos cuando ajustamos los parámetros. Estas métricas nos servirán para evaluar el desempeño de nuestros métodos y la competencia de sus resultados. Similar a lo que ocurre con la función de pérdida, estas métricas son funciones que dependen de la predicción $\widehat{\mathbf{y}}$ y de la etiqueta o verdad fundamental $\mathbf{y}$. Así mismo, cuando nos interese optimizar (maximizar o minimizar) la métrica respecto a un conjunto $\{(\widehat{\mathbf{y}}^{(m)}, \mathbf{y}^{(m)})\}_{m\in\mathcal{M}}$, consideraremos el promedio en este conjunto.

La segmentación puede verse como una tarea de clasificación, donde tenemos que asignarle a cada píxel una clase (aunque la clase del píxel depende de una vecindad del píxel y no solo del píxel por si mismo), por lo que es natural llevar algunas de las métricas de clasificación a la segmentación; por ejemplo, la **accuracy** definida de la siguiente manera:

$$\text{Accuracy} = \frac{|\{(i,j) \mid \mathbf{y}(i,j) = \widehat{\mathbf{y}}(i,j)\}|}{HW}.$$

En ocasiones nos es más sencillo expresar estas fórmulas en términos de las siguientes cantidades

$$\begin{aligned}
\mathrm{TP} &= |\{(i,j) \in \widehat{O} \mid (i,j) \in O\}|, \\
\mathrm{FN} &= |\{(i,j) \in \widehat{O}^c \mid (i,j) \in O\}|, \\
\mathrm{TN} &= |\{(i,j) \in \widehat{O}^c \mid (i,j) \in O^c\}|, \\
\mathrm{FP} &= |\{(i,j) \in \widehat{O} \mid (i,j) \in O^c\}|,
\end{aligned}$$

donde el complemento es tomado respecto a $\{1,...,H\} \times \{1,...,W\}$ y $|\cdot|$ corresponde a la cardinalidad. Estas cantidades son los elementos de la **matriz de confusión**

$$\mathrm{Conf}(\widehat{\mathbf{y}}, \mathbf{y}) = \begin{pmatrix} \mathrm{TP} & \mathrm{FN} \\ \mathrm{FP} & \mathrm{TN} \end{pmatrix},$$

TP es la cantidad de verdaderos positivos ($\widehat{\mathbf{y}} = 1, \mathbf{y} = 1$), FN es la cantidad de falsos negativos ($\widehat{\mathbf{y}} = 0, \mathbf{y} = 1$), TN es la cantidad de verdaderos negativos ($\widehat{\mathbf{y}} = 0, \mathbf{y} = 0$), y finalmente, FP es la cantidad de falsos positivos ($\widehat{\mathbf{y}} = 1, \mathbf{y} = 0$).

Por ejemplo, se tiene

$$\mathrm{Accuracy} = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN}}.$$

Otras métricas utilizadas en clasificación que también se utilizan en segmentación son las siguientes:

$$\begin{aligned}
\mathrm{Precision} &= \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}, \\
\mathrm{Recall} &= \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}},
\end{aligned}$$

y su media armónica denominada **F1-score**,

$$\begin{aligned}
\mathrm{F1} &= \frac{1}{\mathrm{Precision}^{-1} + \mathrm{Recall}^{-1}} \\
&= 2\frac{\mathrm{Precision} \cdot \mathrm{Recall}}{\mathrm{Precision} + \mathrm{Recall}} \\
&= 2\frac{\mathrm{TP}}{2\mathrm{TP} + \mathrm{FN} + \mathrm{FP}}.
\end{aligned}$$

Esta última igualdad se da cuando tanto **Precision** como Recall están bien definidos. Esto no siempre ocurre, así que en este trabajo consideraremos **F1** como la última igualdad.

En el contexto de segmentación nos encontraremos frecuentemente con el **coeficiente de Dice** o **coeficiente de Sørensen-Dice** el cual está definido como

$$\mathrm{DC} = 2\frac{|O \cap \widehat{O}|}{|O| + |\widehat{O}|}.$$

En realidad, esta métrica es la misma que el F1-score, como mostraremos a continuación

**Proposición 3.2.** *La F1-score y el coeficiente de Dice son iguales.*

*Demostración.* Primero, notamos que $|O \cap \widehat{O}| = \mathrm{TP}$ y $|O \cup \widehat{O}| = \mathrm{TP} + \mathrm{FN} + \mathrm{FP}$, luego se sigue que

$$\begin{aligned}
\mathrm{DC} &= 2\frac{|O \cap \widehat{O}|}{|O| + |\widehat{O}|} \\
&= 2\frac{|O \cap \widehat{O}|}{|O \cap \widehat{O}| + |O \cup \widehat{O}|} \\
&= 2\frac{\mathrm{TP}}{\mathrm{TP} + (\mathrm{TP} + \mathrm{FN} + \mathrm{FP})} \\
&= 2\frac{\mathrm{TP}}{2\mathrm{TP} + \mathrm{FN} + \mathrm{FP}} \\
&= \mathrm{F1}.
\end{aligned}$$

□

En general, para cualquier $\beta \in \mathbb{R}^{\geq 0}$ podemos definir un $F_\beta$-score, como

$$F_\beta = \frac{\mathrm{Precision} \cdot \mathrm{Recall}}{(\beta^2 \cdot \mathrm{Precision}) + \mathrm{Recall}}.$$

Mientras más alta sea la $\beta$ más peso le da al *recall* comparado con la *precision*. Esta métrica se utiliza en [23] con $\beta = 2$, sin embargo, nosotros no haremos uso de ella.

Otra métrica que podemos definir, similar al coeficiente de Dice es la intersección sobre la unión (IoU), que como su nombre lo indica está dada por

$$\mathrm{IoU} = \frac{|O \cap \widehat{O}|}{|O \cup \widehat{O}|},$$

que con argumentos similares a los utilizados en la prueba del teorema 3.2, podemos concluir que

$$\text{IoU} = \frac{\text{TP}}{\text{TP} + \text{FN} + \text{FP}}.$$

### 3.5.2. Funciones de pérdida para la segmentación

En esta sección veremos algunas opciones existentes de funciones de pérdida posibles para el problema de segmentación [24–27].

Recordemos que la función de pérdida debe ser una función $L : \mathbb{R}^{H \times W} \to \mathbb{R}$. Al igual que con las métricas, el hecho de que la segmentación pueda ser vista como una clasificación por cada píxel de la imagen nos permite utilizar funciones de pérdida que se emplean para entrenar clasificadores y adaptarlos a este problema.

Cuando entrenamos un clasificador binario $g(\cdot, \boldsymbol{\theta})$, es común que tengamos un modelo cuya salida sean números en el intervalo $[0, 1]$. Es decir, si tenemos la pareja $(\mathbf{x}, y)$ de un punto muestral y su etiqueta, el modelo $g$ estará definido de forma tal que

$$g(\mathbf{x}; \boldsymbol{\theta}) = \widehat{p} \in [0, 1]$$

Es usual que esta salida la interpretemos como la probabilidad $P(y = 1) = \widehat{p}$, y por lo tanto $P(y = 0) = 1 - \widehat{p}$. Notemos que esto es una interpretación y no un hecho comprobable. El modelo del que hablamos anteriormente no es un clasificador *per se*, pues para que lo fuera, la salida debería ser un valor binario $\widehat{y} \in \{0, 1\}$. Entre los métodos más comunes para completar la clasificación luego de obtener las probabilidades, se hace uso de una **umbralización**, donde definimos

$$\widehat{y} = \begin{cases} 1 & \text{si } \widehat{p} \geq T \\ 0 & \text{si } \widehat{p} < T, \end{cases}$$

donde $T \in [0, 1]$. En este caso, la función de pérdida va a ser una función de $\widehat{p}$ en lugar de $\widehat{y}$.

En segmentación, ocurre algo similar. Nuestro modelo debería arrojarnos una salida en $\{0, 1\}^{H \times W}$, sin embargo puede ocurrir que la salida esté en $\mathbb{R}^{H \times W}$ (normalmente en

$[0,1]^{H\times W}$). Así que recurriremos al mismo proceso de umbralización píxel por píxel. Es decir, si obtuvimos el tensor $\widehat{\mathbf{p}}$, definimos mediante umbralización

$$\widehat{y}_{i,j} = \begin{cases} 1 & \text{si } \widehat{p}_{i,j} > T \\ 0 & \text{si } \widehat{p}_{i,j} < T, \end{cases}$$

para todo $i \in \{1,...,H\}$, $j \in \{1,...,W\}$.

Una de las funciones de pérdida más comunes en clasificación es la **Binary Cross Entropy Loss**. Esta función está definida como

$$\text{BCE}(\widehat{p}) = -(y\log(\widehat{p}) + (1-y)\log(1-\widehat{p})).$$

Como hemos estado haciendo con todos los conceptos en clasificación, adaptar esta función al problema de segmentación solo requiere de tomarla píxel por píxel y sumarlas o promediarlas, por ejemplo, tomando el promedio tendríamos

$$\text{BCE}(\widehat{\mathbf{p}}) = \frac{1}{HW}\sum_{i=1}^{H}\sum_{j=1}^{W}\text{BCE}(\widehat{p}_{i,j}).$$

Una desventaja de utilizar la función de pérdida BCE es que cuando existe mucho desbalance de clases entre las muestras *i.e.*, hay muchas más muestras de una clase que de otra no funciona tan bien como otras alternativas. Entre las posibles funciones de pérdida que solucionan este problema, se encuentra la **Weighted Binary Cross Entropy Loss** [26]

$$\text{WBCE}(\widehat{p}) = -(\beta y\log(\widehat{p}) + (1-y)\log(1-\widehat{p})).$$

Escoger $\beta > 1$ ayuda a reducir el número de falsos negativos, de la misma forma, escoger $\beta < 1$ ayuda a reducir el número de falsos positivos.

El coeficiente de Dice, puede ser modificado para ser una función de pérdida a la cual denominaremos **Dice Loss**.

$$\text{DL}(\widehat{\mathbf{p}}) = 1 - 2\frac{\sum_{i=1}^{H}\sum_{j=0}^{W}\widehat{p}_{i,j}y_{i,j} + \epsilon}{\sum_{i}^{H}\sum_{j}^{W}(\widehat{p}_{i,j} + y_{i,j}) + \epsilon},$$

donde $\epsilon > 0$ está para evitar divisiones entre 0. Notemos que lo que hay en el numerador sería la intersección $|O \cap \widehat{O}|$ si en lugar de $\widehat{p}_{i,j}$, tuvieramos $\widehat{y}_{i,j}$, y así mismo el denominador

sería $|O| + |\widehat{O}|$. Para aumentar la velocidad de convergencia podemos modificar la función elevando los sumandos del denominador al cuadrado de la siguiente manera:

$$\mathrm{DL}(\widehat{\mathbf{p}}) = 1 - 2\frac{\sum_{i=1}^{H}\sum_{j=0}^{W}\widehat{p}_{i,j}y_{i,j} + \epsilon}{\sum_{i}^{H}\sum_{j}^{W}\left(\widehat{p}_{i,j}^{2} + y_{i,j}^{2}\right) + \epsilon}.$$

# Capítulo 4

# Aprendizaje Profundo

Vimos en el Capítulo 3.1 algunos conceptos fundamentales en el aprendizaje automático. Uno de los factores primordiales a la hora de resolver un problema, de aprendizaje automático, es escoger el modelo correcto. Existen distintos modelos que se adaptan bien a distintos tipos de problemas, sin embargo, nosotros nos enfocaremos en una familia de modelos que ha demostrado ser útil en una amplia gama de problemas, las **redes neuronales convolucionales** (**CNN**). Cuando contamos con muchas capas estamos en presencia del subdominio del aprendizaje de máquina conocido como **aprendizaje profundo** (**DL**). Estos modelos han revolucionado el área, superando a sus predecesores en la solución de ciertos problemas por un amplio margen. Estos modelos son útiles cuando trabajamos con información dispuesta en forma reticular, como lo son las imágenes. Como ejemplo de estas redes tenemos la ResNet [16], de la cuál hablaremos más adelante, la cuál el 2015 logró conseguir una error del 3.57 % en el ImageNet-2012, superando a la capacidad humana (error del 5 %); el ImageNet es un reto en el cual se debían clasificar 150,000 imágenes etiquetadas en más de 10,000 posibles clases. Otro hito conseguido por el DL es vencer al 18 veces campeón del mundo de Go, Lee Sedol, en este juego en el año 2016 [28].

## 4.1. Redes Neuronales Convolucionales (CNN)

Las CNN no tienen una forma tan específica como otros modelos, sino que más bien se componen de distintas operaciones con propósitos específicos que en conjunto nos ayudan a resolver un problema general. A estas operaciones les llamaremos **bloques** en este contexto y a un modelo en particular le llamaremos una **arquitectura**, con la intuición de que podemos tomar varios bloques para construir una arquitectura de manera modular. Algunos de estos bloques contienen parámetros (que deben ser entrenados) y otros no.

### 4.1.1. Convoluciones

Los bloques principales, y a su vez los que definen una CNN son las **convoluciones**, estos requieren de un **filtro** o **kernel**, que es una matriz $\mathbf{k} \in \mathbb{R}^{C\times F\times F}$, con $F$ impar (Es posible definir filtros rectangulares y con dimensiones pares, pero ya que no es muy común, no los tomaremos en cuenta). Definimos entonces $\text{Conv} : \mathbb{R}^{C\times H\times W} \to \mathbb{R}^{1\times(H-2R)\times(W-2R)}$ como

$$\text{Conv}(\mathbf{x};\mathbf{k})_{i,j} = \sum_{c}^{C}\left(\sum_{h,w=-R}^{R} k_{c,(h+R+1),(w+R+1)} x_{c,(i+h),(j+w)}\right),$$

donde $R = (F-1)/2$.

Al igual que en la regresión lineal, en las CNN es importante definir un sesgo $b \in \mathbb{R}$. De manera que de hecho las convoluciones que usaremos son de la forma

$$\text{Conv}(\mathbf{x};\mathbf{k},b)_{i,j} = \text{Conv}(\mathbf{x};\mathbf{k})_{i,j} + b,$$

Como última consideración respecto a las convoluciones, notamos que el contradominio de Conv es de la forma $\mathbb{R}^{1\times H'\times W'}$, es decir que la salida tiene un solo canal, independientemente de cuantos canales tenga la entrada. Para conseguir que la salida tenga más canales, digamos $D$, podemos aplicar $D$ convoluciones y concatenarlos en la dimensión de los canales

$$\text{Conv}(\mathbf{x};\mathbf{K},\mathbf{b})_{d,i,j} = \text{Conv}(\mathbf{x};\mathbf{k}_d,b_d)_{i,j},$$

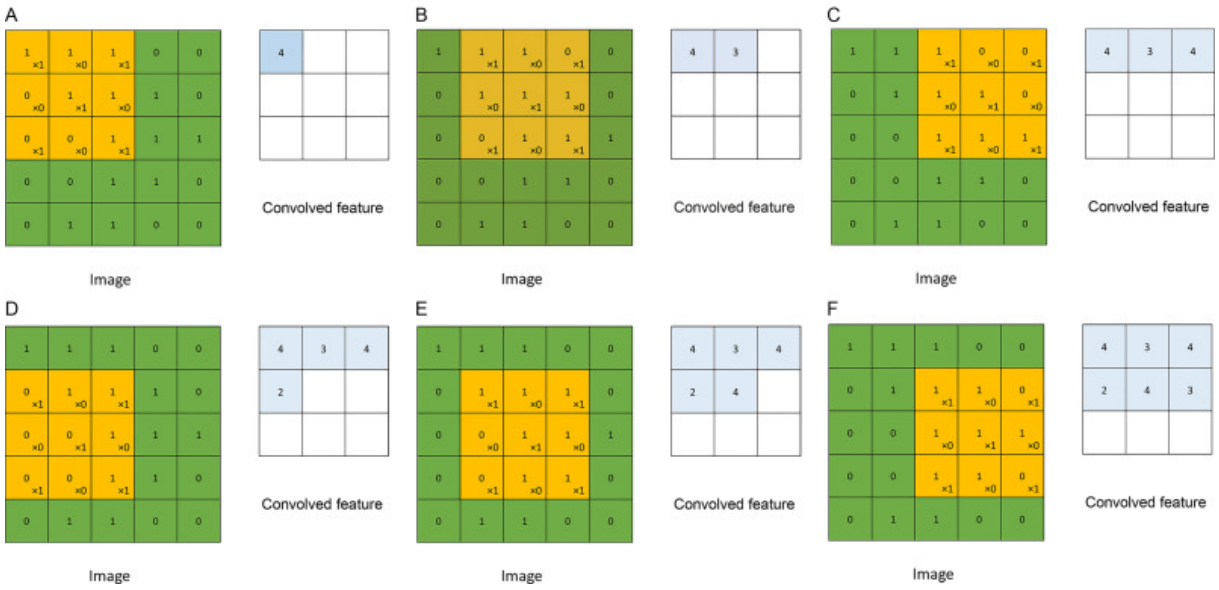



Figura 4.1: Convolución de una imagen con un kernel. Imagen extraída de [29].

donde $\mathbf{K} = \{\mathbf{k}_1, ..., \mathbf{k}_D\}$ y $\mathbf{b} = \{b_1, ..., b_D\}$.

Es posible interpretar las convoluciones en dos dimensiones como que el kernel se mueve a lo largo y ancho de la imagen realizando un producto puntual y sumando los resultados. Esta interpretación se puede ver en la Figura 4.1.

Por último, para que el modelo pueda aprender las no linealidades del problema, es necesario que después de una capa convolucional, apliquemos una función de activación no lineal $\sigma$.

Las convoluciones no son sensibles a traslaciones, y utilizan una cantidad reducida de parámetros en comparación de su predecesor, las redes neuronales artificiales [30]. Además, las operaciones se hacen en vecindarios de los píxeles, por lo cual aprende bien las características locales. Esto combinado con la siguiente operación que veremos constituye la base de las CNN y les dota de la habilidad de capturar patrones complejos en diferentes escalas y posiciones de una imagen.

### 4.1.2. Convolución con paso

Una convolución con un $S$-paso ($S$-*stride*), la definiremos como $\text{Conv}_s(\cdot;\mathbf{k}) : \mathbb{R}^{C\times H\times W} \to \mathbb{R}^{1\times\lceil H/S\rceil\times\lceil W/S\rceil}$

$$\text{Conv}_S(\mathbf{x};\mathbf{k})_{i,j} = \sum_{c}^{C}\left(\sum_{h,w=-R}^{R} k_{c,(h+R+1),(w+R+1)} x_{c,(ai-a+1+h),(aj-a+1+w)}\right),$$

donde $R = (F-1)/2$.

Y definimos análogamente al caso de las convoluciones $\text{Conv}_S(\mathbf{x};\mathbf{k},b)$ cuando incluimos el sesgo y $\text{Conv}(\mathbf{x};\mathbf{K},\mathbf{b})$ en $\mathbb{R}^{D\times\lceil H/S\rceil\times\lceil W/S\rceil}$ cuando realizamos $D$ convoluciones con paso.

### 4.1.3. Funciones de activación

Mencionamos en la Subsección 4.1.1, que seguido de una convolución era necesario aplicar una función de activación $\sigma$ no lineal. Queremos que $\sigma$ cumpla las siguientes condiciones

1. $\sigma$ debe ser no afín.

2. $\sigma$ debe ser derivable en casi todas partes.

3. Preferiblemente $\sigma$ debe ser computacionalmente fácil de calcular.

El primer punto es importante porque las convoluciones son funciones afines y la composición de funciones afines es afín. En general las funciones que modelan la realidad son más complejas que una función lineal, por lo tanto es importante que la red pueda aprender las no linealidades del problema a resolver. La diferenciabilidad de $\sigma$ será importante cuando realicemos el entrenamiento de la red y que sea fácil de calcular es útil para que el modelo no tome tanto tiempo en entrenarse.

Las opciones más comunes para escoger como función de activación son:

1. Tangente hiperbólica.

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}.$$

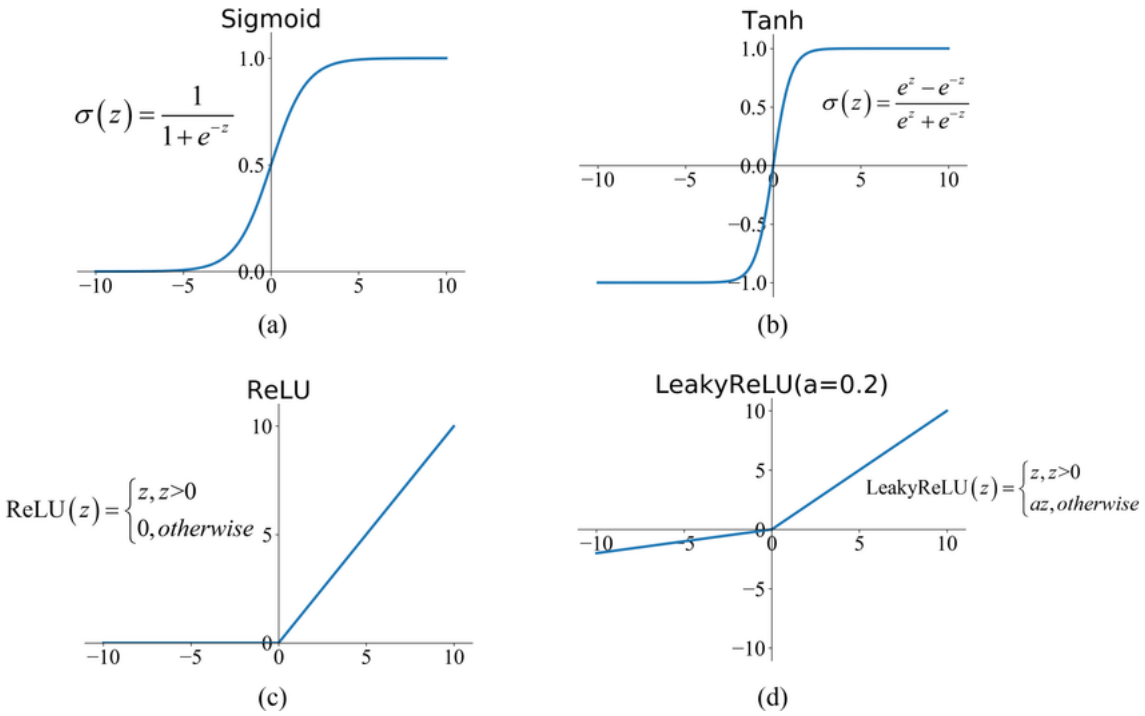



Figura 4.2: Gráficas de diferentes funciones de activación.

2. Sigmoide:

$$\sigma(x) = \frac{1}{1+e^{-x}}.$$

3. Unidad linear rectificadora (ReLU):

$$\sigma(x) = \text{máx}(0, x).$$

4. Leaky ReLU:

$$\sigma(x) = \begin{cases} \text{máx}(0,x) & \text{si } x > 0 \\ 0.01x & \text{si } x \leq 0. \end{cases}$$

La Figura 4.2 muestra éstas funciones de activación.

### 4.1.4. Relleno

Las convoluciones regulares reducen la altura y el ancho de la imagen, sin embargo hay ocasiones en que queremos evitar este comportamiento. Para esto, extendemos la imagen

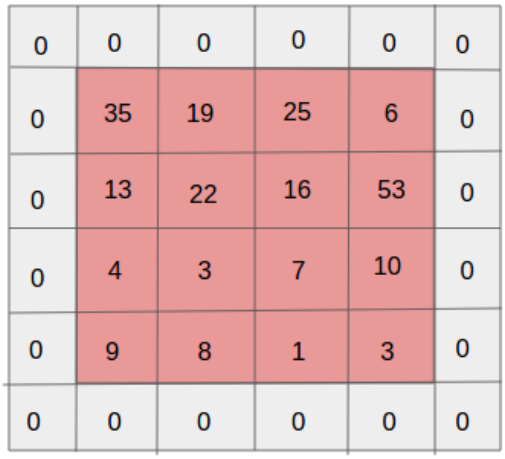



Figura 4.3: Relleno de ceros aplicado a una imagen, la parte roja corresponde a la imagen original. Imagen extraída de [31].

a dimensiones $C \times (H + 2E) \times (W + 2E)$, a esto se le llama **relleno** (*padding*). El método que utilizaremos en las redes de este trabajo se llama **relleno de ceros** y se define

$$\text{Pad}_E(\mathbf{x})_{i,j} \begin{cases} \mathbf{x}_{i-E,j-E} & \text{si } E+1 \leq i \leq H-E, E \leq j \leq W-E \\ 0 & \text{en otro caso,} \end{cases}$$

(ver Figura 4.3). En particular, tomamos $E = R$ (recordemos que $R = (F-1)/2$) cuando queremos hacer una convolución y conservar el tamaño original. Normalmente consideramos el relleno en la misma operación de la convolución.

### 4.1.5. Submuestreo

El **submuestreo** o **pooling** es una operación cuya salida reduce las dimensiones (alto y ancho) del tensor. Si únicamente utilizáramos convoluciones en nuestro modelo, por su naturaleza local, no podría aprender correlaciones entre píxeles fuera del rango que abarca el kernel. Además, mantener las mismas dimensiones es costoso computacionalmente. Es por esto que un componente básico en las CNN es el submuestreo. Estas operaciones, al igual que las convoluciones, las podemos interpretar como que un kernel de dimensiones

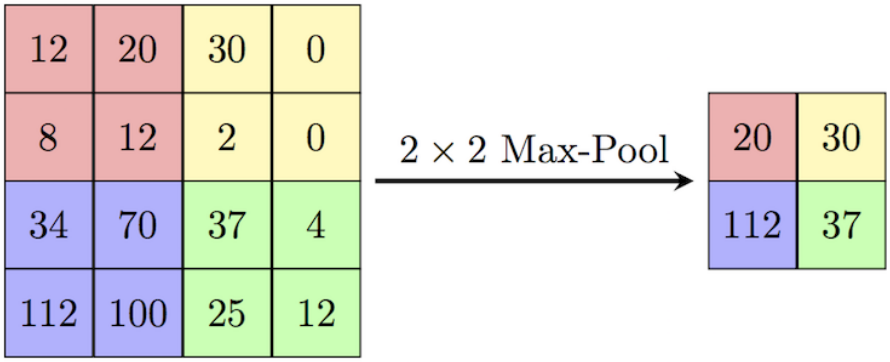



Figura 4.4: Max-pooling con $P = 2$, $S = 2$. Imagen extraída de [32].

$P \times P$ recorre la imagen, y de nuevo, análogo a como sucede en las convoluciones, también se puede hacer con un salto o paso $S$, de manera tal que si tomamos una entrada $\mathbf{x}$ de dimensiones $C \times H \times W$, realizar el pooling nos devolvería una salida de dimensiones $C \times (\lfloor (H-P)/S \rfloor + 1) \times (\lfloor (W-P)/S \rfloor + 1)$. Lo importante de esta operación es que ayuda a conservar la información contenida en $\mathbf{x}$ de manera local cuando realizamos el entrenamiento. Definimos el conjunto

$$\mathcal{P}_{i,j} = \{x_{i',j'} \mid 1 + (i-1)S \leq i' \leq 1 + (i-1)S + P, 1 + (j-1)S \leq j' \leq 1 + (j-1)S + P\},$$

con éste definimos $\text{Pool}_{S,P} : \mathbb{R}^{H \times W} \to \mathbb{R}^{(\lfloor (H-P)/S \rfloor + 1) \times (\lfloor (W-P)/S \rfloor + 1)}$ como

$$\text{Pool}_{S,P}(\mathbf{x})_{i,j} = a(\mathcal{P}_{i,j}),$$

donde $a$ es la función de agregación que mencionamos previamente. Actualmente la función de agregación más usada es el máximo y en ese caso le denominamos **max-pooling**; la Figura 4.4 muestra un Max-Pooling. Lo más usual, y lo que usamos en nuestros modelos es tomar $S = P$, de forma que la salida tiene dimensiones $\lfloor W/P \rfloor \times \lfloor W/P \rfloor$.

Cuando tenemos $C$ canales realizamos el pooling en cada canal por separado. Es decir, supongamos que tenemos una imagen $\mathbf{x} \in \mathbb{R}^{C \times H \times W}$, digamos que $\mathbf{x}_c \in \mathbb{R}^{H \times W}$ es la imagen en el canal $c$. Definimos entonces

$$\text{Pool}(\mathbf{x})_{c,i,j} = \text{Pool}(\mathbf{x}_c)_{i,j}.$$

**Observación 4.1.** *Estamos abusando de la notación pues la función* Pool *del lado izquierdo tiene como dominio* $\mathbb{R}^{C\times H\times W}$ *y la de la derecha tiene como dominio* $\mathbb{R}^{H\times W}$ *que es el que definimos con anterioridad.*

Una secuencia muy común en una arquitectura de una red neuronal convolucional es la siguiente: una convolución, una función de activación y un pooling.

### 4.1.6. Capas completamente conectadas

Una **capa completamente conectada** o **capa densa** es una capa que realiza una operación afín $\text{Dense} : \mathbb{R}^{N_1} \to \mathbb{R}^{N_2}$, definida por

$$\text{Dense}(\widetilde{\mathbf{x}}; \mathbf{W}, \mathbf{b}) = \mathbf{W}\widetilde{\mathbf{x}} + \mathbf{b},$$

donde $\mathbf{W} \in \mathbb{R}^{N_2\times N_1}$, $\mathbf{b} \in \mathbb{R}^{N_2}$.

Notemos que esta operación es esencialmente diferente al resto que hemos visto, ya que su dominio no tienen estructura reticular. Lo que se suele hacer es "desdoblar" o "aplanar" el tensor $\mathbf{x} \in \mathbb{R}^{C\times H\times W}$ en un vector $\widetilde{\mathbf{x}} \in \mathbb{R}^{CHW}$ para posteriormente aplicarle las capas densas que sean requeridas. Estas capas se ponen comúnmente al final de la arquitectura y normalmente son usadas para clasificación. En este trabajo, no se usarán este tipo de capas.

### 4.1.7. Sobremuestreo

El **sobremuestreo** es lo contrario al submuestreo. Es decir, que si tenemos un tensor en $\mathbb{R}^{C\times H\times W}$, queremos incrementar alto y el ancho de la imagen por un factor $U$; es decir, el tensor resultante del sobremuestreo debe estar en $\mathbb{R}^{C\times HU\times WU}$. Empezaremos considerando el caso de tensores en dos dimensiones; es decir, en $\mathbb{R}^{H\times W}$ debido a que al igual que con el submuestreo (pooling), el hacer sobremuestreo con varios canales sólo es hacer sobremuestreo en cada canal por separado. Existen varias maneras de realizar un sobremuestreo, veremos algunas importantes.

El sobremuestreo normalmente irá precedido por algunas convoluciones y submuestreo, y queremos recuperar las características que perdimos cuando se realizó el submuestreo, de manera que la idea de localidad no se pierde del todo en este tipo de operación.

El primer ejemplo de sobremuestreo es el de **vecinos más cercanos**, donde definimos

$$\mathrm{NearestUp}(\mathbf{x})_{i,j} = \mathbf{x}_{\lceil i/P \rceil, \lceil j/P \rceil}.$$

También, podemos definir el método de "**cama de agujas**", que está dado por la fórmula

$$\mathrm{BedOfNeilsUp}(\mathbf{x})_{i,j} = \begin{cases} \mathbf{x}_{\lceil i/P \rceil, \lceil j/P \rceil} & \text{si } i, j \equiv 1 \pmod{U} \\ 0 & \text{de otra forma.} \end{cases}$$

Por último veremos un método que ayuda a suavizar la salida del sobremuestreo, el **sobremuestreo bilineal**. Para entender este método, primero explicaremos lo que es **interpolación lineal**. Supongamos que tenemos $M$ parejas de puntos, digamos $\{(x_i, y_i)\}_{i=1}^{M} \subset \mathbb{R}^2$. Y queremos encontrar una función $g : \mathbb{R} \to \mathbb{R}$ tal que $g(x_m) = y_m$. Para conseguirlo podemos usar lineas, es decir, definimos las lineas $g_i : [x_i, x_{i+1}] \to \mathbb{R}$

$$g_i(x) = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(x - x_i) + y_i,$$

y definimos $g(x) = g_i(x)$ cuando $x_i \leq x \leq x_{i+1}$ (notemos que está bien definido, puesto que $g_i(x_{i+1}) = g_{i+1}(x_{i+1})$). Por supuesto, falta definir que sucede cuando $x < x_1$ y $x > x_M$, en nuestro caso no nos es útil esta sección de la recta real pero para definirlo continuo puede tomarse por ejemplo $g(x) = g_1(x)$ si $x < x_1$ y $g_{M-1}(x)$ si $x > x_M$.

En ocasiones querremos interpolar puntos en el plano dispuestos en una retícula $\{((x_i, y_j), z_{i,j})\}_{i=1,j=1}^{M,N} \subset \mathbb{R}^2 \times \mathbb{R}$. De forma análoga, queremos encontrar $g : \mathbb{R}^2 \to \mathbb{R}$ tal que $g(x_i, y_j) = z_{i,j}$. El método que nos concierne es la **interpolación bilineal**, que no es más que la aplicación de una interpolación lineal en el eje $X$ y una en el eje $Y$, es decir, para $j = 1, ..., N$ definimos las $M$ lineas $g_{ji}$ como

$$g_{ji}(x) = \frac{z_{(i+1),j} - z_{i,j}}{x_{i+1} - x_i}(x - x_i) + z_{i,j},$$

con lo que podemos definir $g(x, y_j) = g_{ji}(x)$ si $x_i \leq x \leq x_{i+1}$. Ahora, para toda $x$ definimos las lineas $g_{x,j}$

$$g_{x,j}(y) = \frac{g_{(j+1),i}(x) - g_{j,i}(x)}{y_{j+1} - y_j}(y - y_j) + g_{j,i}(x),$$

y definir $g(x, y) = g_{x,j}(y)$ si $y_j \leq y \leq y_{j+1}$.

Resulta que interpolar linealmente primero en el eje $X$ y luego en el eje $Y$ es lo mismo que interpolar linealmente en el eje $Y$ y luego en el eje $X$.

El siguiente paso es utilizar este método para realizar sobremuestreo, o sea, realizar una interpolación en un espacio discreto. Esto se puede realizar de distintas formas, veremos un ejemplo que está implementado en PyTorch. Ya que interpolar bilinealmente es lo mismo que realizar dos interpolaciones lineales, describiré el método para una interpolación lineal. Sea $\mathbf{x} \in \mathbb{R}^N$ una imagen en escala de grises, queremos definir una función $\text{LinearUp} : \mathbb{R}^N \to \mathbb{R}^{N'}$ basada en interpolación lineal. Interpolaremos respecto a las parejas

$$\left\{\left(\frac{i-1}{N-1}, x_i\right)\right\}_{i=1}^{M}.$$

A la función interpoladora le llamamos $g$ y definimos

$$\text{LinearUp}(\mathbf{x})_{i'} = g\left(\frac{i'-1}{N'-1}\right).$$

De manera similar, en el caso bidimensional, cuando contamos con una imagen $\mathbb{R}^{H \times W}$ podemos definir $g$ como la interpolación respecto a

$$\left\{\left(\left(\frac{i-1}{H-1}, \frac{j-1}{W-1}\right), x_{ij}\right)\right\}_{i=1,j=1}^{H,W},$$

y definir $\text{BilinearUp} : \mathbb{R}^{H \times W} \to \mathbb{R}^{H' \times W'}$

$$\text{BilinearUp}(\mathbf{x})_{i',j'} = g\left(\frac{i'-1}{H'-1}, \frac{j'-1}{W'-1}\right).$$

Como es costumbre, nos interesa definir una función más general $\text{BilinearUp} : \mathbb{R}^{C \times H \times W} \to \mathbb{R}^{C \times H' \times W'}$, y de nuevo esto se hace aplicando la función que ya definimos independientemente en cada canal.

De hecho, no necesariamente $H' > H$ ni $W' > W$, cuando tengamos $H' < H$ y $W' < W$, a este método le llamaremos **submuestreo bilineal**.

### 4.1.8. Convoluciones Transpuestas

Las **convoluciones transpuestas** (también llamadas deconvoluciones), son utilizadas para realizar sobremuestreo, pero de forma paramétrica, es decir, que una convolución transpuesta contiene pesos que entrenar. Estos se utilizan en super-resolución de imágenes y en segmentación. En particular, la U-Net que veremos más adelante, requiere de sobremuestreo y algunas variantes implementan convoluciones transpuestas en lugar de los otros métodos que hemos discutido.

La convolución (de $\mathbb{R}^{H\times W} \to \mathbb{R}^{(H-2R)\times(W-2R)}$), como hemos visto es una operación lineal, así que se puede ver como

$$\mathrm{Conv}(\mathbf{x};\mathbf{k}) = \mathbf{W}\mathrm{Vec}(\mathbf{x}),$$

donde $\mathbf{W}$ es una matriz conformada por las entradas de $\mathbf{k}$ y $\mathrm{Vec}(x)$ es la matriz $\mathbf{x}$ "aplanada", de manera que $\mathrm{Vec}(\mathbf{x}) \in \mathbb{R}^{HW}$. De la misma manera, podemos definir una operación $\mathrm{TransConv} : \mathbb{R}^{H-2R\times W-2R} \to \mathbb{R}^{H\times W}$, como

$$\mathrm{TransConv}(\mathbf{x};\mathbf{k}) = \mathbf{W}^\top\mathrm{Vec}(\mathbf{x}).$$

También podemos hacer una extensión donde el dominio y el contradominio tengan más de un canal y una versión donde definamos el paso, como en las convoluciones convencionales; esto último nos sirve para hacer el sobremuestreo.

### 4.1.9. Regularización

Para evitar el sobreajuste al igual que en la regresión lineal, se puede agregar un término $R(\boldsymbol{\theta})$ a la función de pérdida, sin embargo es más común por la eficiencia computacional realizar alguno de los siguientes métodos.

#### Abandono (Dropout)

El **abandono** es una técnica que se utiliza durante el entrenamiento y consiste en "apagar" algunos pesos con cierta probabilidad en cada iteración. Supongamos que tengo

una capa convolucional con parámetros $\mathbf{K}$. Agregarle abandono a esta capa significa que durante el entrenamiento en cada época, con probabilidad $p \in [0, 1)$ cada peso $K_{i,j}$ va a ser 0 en la propagación hacia adelante. Todos los demás pasos del entrenamiento de una red neuronal proceden igual. Esto permite que aprenda patrones sin depender de ciertos pesos específicos.

### Normalización por lotes (*Batch Normalization*)

Recordemos que en ocasiones vamos a optimizar por lotes, en estos casos, podemos aplicar normalización por lotes [33]. Supongamos que tenemos el lote $\{\mathbf{x}^{(m_1)}, ..., \mathbf{x}^{(m_K)}\}$, entonces definimos

$$\text{BatchNorm}(\mathbf{x}^{(m_k)}; \gamma, \beta) = \frac{\gamma(\mathbf{x}^{(m_k)} - \mu)}{\sigma} + \beta,$$

donde

$$\mu = \sum_{k=1}^{K} \frac{\mathbf{x}^{(m_k)}}{K}, \quad \sigma^2 = \sum_{k=1} \frac{(\mathbf{x}^{(m_k)} - \mu)^2}{K}.$$

Aplicar normalización por lotes durante el entrenamiento reduce la necesidad de utilizar abandono. La normalización por lotes se aplica antes de la función de activación. Cabe recalcar que los parámetros $\gamma$ y $\beta$ se aprenden en el entrenamiento.

### Aumento de datos

El caso ideal para evitar el sobreajuste, es que la red tenga suficientes muestras para aprender, sin embargo esto no siempre es posible. Cuando trabajamos con imágenes, es posible simular más muestras aplicando transformaciones a las muestras que tenemos y tomándolas como muestras adicionales. Por ejemplo aplicar rotaciones, traslaciones, reflexiones, cambios de color, escalamientos, etc.

En el caso de la segmentación, siempre que generemos una nueva muestra de una transformación espacial, también se debe generar la etiqueta mediante la misma transformación. Las Figuras 4.5 y 4.6 muestran un ejemplo de aumento de datos, en particular la Figura 4.6 ejemplifica el aumento de datos en el caso de la segmentación.

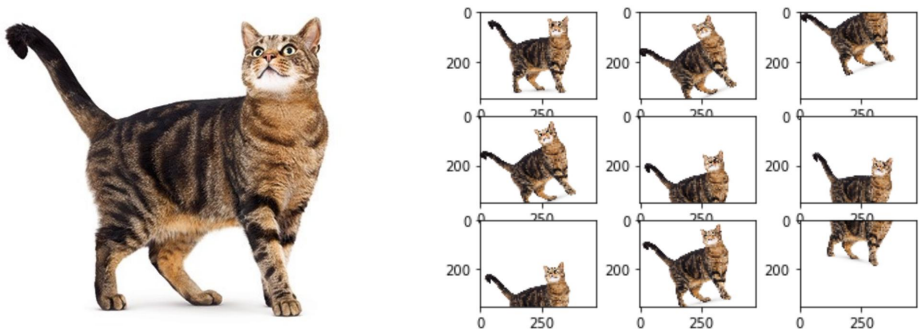


Figura 4.5: Aumento de datos, a la izquierda se puede apreciar una muestra que entraría en el modelo, y a la derecha posibles transformaciones que se le realizarían al gato. Todas estas nuevas imágenes formarían parte de nuestro conjunto de datos extendido que utilizaremos para entrenar la red. Imagen extraída de [34].

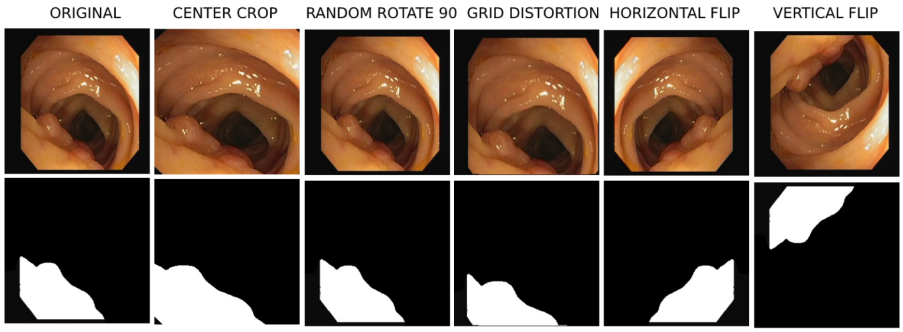



Figura 4.6: Aumento de datos en segmentación. Se puede ver que también se generan las etiquetas con las mismas transformaciones que se usan en la imagen. Imagen extraída de [35].

### 4.1.10. Inicialización de pesos

Inicializar los pesos negligentemente puede llevar al problema del desvanecimiento del gradiente y retrasar o impedir la convergencia [36]. Debido a esto, se han propuesto diversos métodos para inicializar los pesos de una red neuronal.

Lo más natural es inicializar los pesos respecto a una distribución normal o respecto a una distribución uniforme y de hecho, los métodos estándar utilizados actualmente inicializan los parámetros de esta manera.

Hemos abarcado únicamente tres tipos de operaciones paramétricas: convoluciones, convoluciones transpuestas y capas densas. Lo que tienen en común estas capas es que todas son operaciones afines. Así que para esta discusión, supondremos que tenemos una capa

$$f(\mathbf{x}; \mathbf{W}, \mathbf{b}) = \mathbf{W}\mathbf{x} + \mathbf{b}.$$

con $\mathbf{W} \in \mathbb{R}^{N \times \widetilde{N}}$. Tanto en las convoluciones como en las convoluciones transpuestas $\mathbf{x}$ representa el tensor convertido en vector, por ejemplo, en las convoluciones $\mathbf{x} \in \mathbb{R}^{F^2C}$ y $N = F^2C$. $\widetilde{N}$ es el tamaño del vector de salida. A esta operación afín le sucede una función de activación $\sigma$.

En [37], Xavier Glorot bajo la suposición a priori de que las entradas de $\mathbf{x}$ y de $\mathbf{W}$ son independientes y están idénticamente distribuidas, propone inicializar $\mathbf{b} = 0$ y las entradas de $\mathbf{W}$ tomadas de una distribución normal o uniforme con media 0 y varianza

$$\frac{2}{N + \widetilde{N}}.$$

Este método es conocido como **inicialización de Xavier** o **Inicialización de Glorot**.

Kaming He en [38], propone un método similar al de Glorot, tomando la suposición adicional de que $\sigma$ es la función ReLu. Con la consideración adicional, se propone tomar una distribución (normal o uniforme) con media 0 y varianza

$$\frac{2}{N}.$$

A esta manera de inicializar los pesos se le conoce como **inicialización de Kaming**, por defecto PyTorch utiliza este método con una distribución normal.

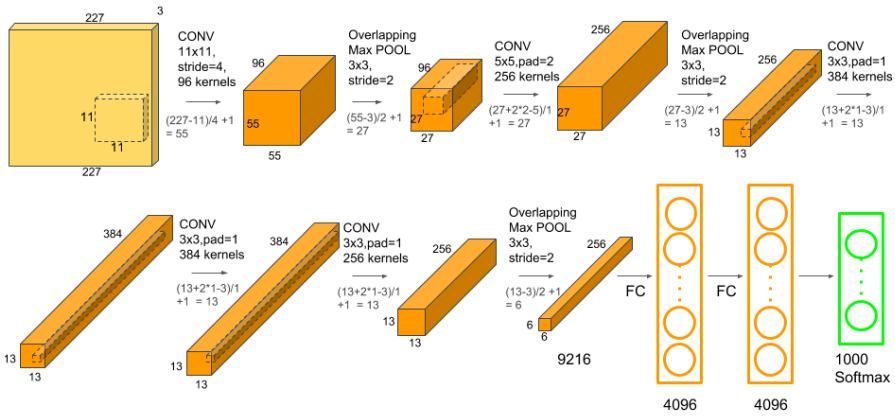



Figura 4.7: Diagrama de la AlexNet, Imagen extraída de [39].

.

Para finalizar esta sección, en la Figura 4.7, podemos observar como se integran muchos de los elementos introducidos en una red neuronal profunda, la AlexNet.

## 4.2. Aprendizaje de las CNN

Por más complejas que puedan ser las capas que hemos visto (convoluciones, convoluciones transpuestas, etc.), las arquitecturas compuestas por éstas no dejan de ser modelos paramétricos $\widehat{f}(\cdot;\boldsymbol{\theta})$, en donde queremos encontrar los valores óptimos para $\boldsymbol{\theta}$. Así que como vimos en el Capítulo 3.1, una de las formas más comunes para hacer esto es calculando $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ y utilizando algún algoritmo como el descenso de gradiente. Pese a que esencialmente el problema se reduce a esto, calcular $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ puede suponer una tarea compleja para modelos como estos. Claramente no podemos calcular analíticamente el gradiente como en nuestro ejemplo pequeño del capítulo 3.1. La alternativa más común es utilizar el algoritmo de **propagación hacia atrás** o **retropropagación**.

Este algoritmo está basado en el **teorema de la cadena**. En su versión univariable

dice que si $y = f(x)$ y $z = g(y)$, donde $x, y, z \in \mathbb{R}$, entonces

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}.$$

Este teorema se puede generalizar para más dimensiones, de manera que si $\mathbf{y} = f(\mathbf{x})$ y $z = g(\mathbf{y})$, donde $\mathbf{x} \in \mathbb{R}^N$, $\mathbf{y} \in \mathbb{R}^M$ y $z \in \mathbb{R}$, entonces

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}.$$

Para implementar el algoritmo, conforme se van realizando operaciones se va creando un grafo, de hecho, un árbol el cual utilizaremos junto con el teorema de la cadena para calcular el gradiente.

Aunque el algoritmo del descenso de gradiente escala bien y puede ser utilizado para actualizar los pesos, es más común utilizar algoritmos más robustos. Nosotros utilizaremos el ADAM [18], que ha demostrado ser un buen optimizador por defecto.

## 4.3. Redes Neuronales Residuales (ResNet)

El éxito de las redes neuronales se acrecentó conforme la profundidad de éstas iba aumentando. Uno pensaría que con suficientes muestras y suficiente profundidad se podrían alcanzar resultados arbitrariamente buenos, sin embargo, además del costo y por tanto, tiempo computacional que implica entrenar a una red de estas proporciones, hay otro problema a la hora incrementar la profundidad: **el problema de desvanecimiento del gradiente**. Este problema, como su nombre lo indica, refiere a que en estas redes profundas, el gradiente de la función de pérdida con respecto a ciertos parámetros se hace cero, y esto implica que estos pesos no se actualizarán, provocando a su vez que no continúe el aprendizaje correctamente. Una de las causas del problema es la manera en la que encontramos el gradiente, mediante propagación hacia atrás. Ya que estamos multiplicando una gran cantidad de números, cuando estos se vuelven muy pequeños el error numérico causado por la aritmética de punto flotante provoca que el gradiente que calculamos sea cero.

En diciembre del 2015, Kaming He y sus coautores, proponen una arquitectura que soluciona este problema, permitiendo el desarrollo de redes más profundas. En [16], por ejemplo construyen arquitecturas de 101 y 152 capas. La idea detrás de estas redes es simple, y lleva por nombre conexiones de salto; las redes que utilizan estas conexiones reciben el nombre de **redes residuales**.

Las conexiones de salto funcionan de la siguiente manera: supongamos que tenemos una pequeña red $f_1(\cdot;\boldsymbol{\theta}_1)$ (por ejemplo convolución, activación, pooling). En una red convencional podríamos aplicar otra pequeña red, digamos $f_2(\cdot;\boldsymbol{\theta}_2)$, de manera que la siguiente salida sería $f_2(f_1(\mathbf{x};\boldsymbol{\theta}_1);\boldsymbol{\theta}_2)$. En una ResNet, en su lugar, tenemos como salida $f_2(\mathbf{x} + f_1(\mathbf{x};\boldsymbol{\theta}_1);\boldsymbol{\theta}_2)$. A estas funciones $f_1$ y $f_2$ les llamamos **bloques residuales** (**ResBlock**). Es decir, si tenemos $S$ ResBlocks, tendremos la siguiente fórmula recursiva

$$\mathbf{z}_{s+1} = \mathbf{z}_s + f_s(\mathbf{z}_s;\boldsymbol{\theta}), \quad \mathbf{z}_0 = \mathbf{x}, \tag{4.1}$$

que nos permite calcular la salida $\mathbf{y} = \mathbf{z}_S$ (En ocaciones después de calcular $\mathbf{z}_S$, realizaremos más operaciones no residuales para encontrar la salida, en ese caso la salida sería $\mathbf{y} = \sigma(\mathbf{z}_S;\boldsymbol{\phi})$, donde $\sigma$ son estas operaciones adicionales). Dado que el algoritmo de propagación hacia atrás calcula el gradiente de la función con respecto a los parámetros. Creando conexiones de salto, creamos conexiones más directas de los parámetros más profundos de la red.

Podemos ver la representación gráfica de una conexión de salto en la Figura 4.8

La ResNet, y en particular la ecuación (4.1) son un elemento fundamental en la motivación de la arquitectura que utilizaremos en este trabajo. En la Figura 4.9, se aprecia un ejemplo de ResNet.

## 4.4. CNN en segmentación

En [40], se propone utilizar una red neuronal completamente convolucional para segmentación. Requerimos que la salida $\widehat{\mathbf{y}}$ tenga las mismas dimensiones (alto y ancho) que la verdad fundamental $\mathbf{y}$, sin embargo las convoluciones naturalmente tienden a reducir el

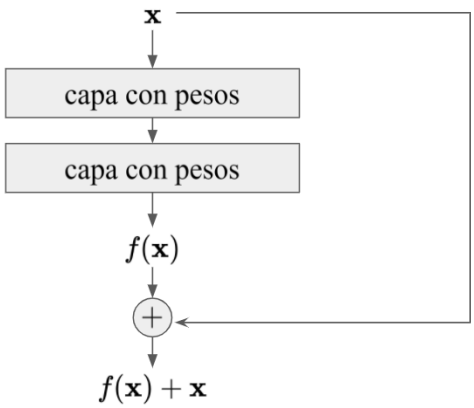



Figura 4.8: Esquema básico de las conexiones de salto. Imagen basada en [16].

tamaño de los tensores conforme se avanza en la profundidad de la red. Entre las estrategias para solucionar esta problemática se encuentran: aplicar relleno a todas las capas convolucionales, de manera que el tensor nunca reduzca el ancho ni el alto. La desventaja de esta estrategia es el costo computacional que conlleva. Otra manera de solucionar este problema es que en la última capa se realice un sobremuestreo (normalmente una convolución transpuesta). Los inconvenientes de este método, son que estamos incrementando abruptamente el tamaño del tensor y podríamos no tener una segmentación muy fina. Para solucionar las problemáticas del método anterior, podemos diseñar una arquitectura en donde las primeras capas reduzcan el tamaño de la entrada gradualmente (mediante submuestreo o convoluciones con paso), a esta parte de la red se le conoce como **codificador**, y la segunda mitad de la red, el **decodificador**, aumente gradualmente el tamaño del tensor, de manera que las dimensiones de la predicción coincidan con las de la etiqueta. A estas arquitecturas se les denomina **codificador-decodificador** (**ED**). El hacer un sobremuestreo gradual, ayuda a que haya menos pérdida de información en la segmentación final, sin embargo hay técnicas que contribuyen a mitigar esta pérdida de información y consecuentemente obteniendo una segmentación más fina. En particular, la siguiente arquitectura que veremos, la **U-Net**, utiliza conexiones de salto para realizar esta tarea. En la Figura 4.10, podemos observar una red completamente convolucional, en particular, un

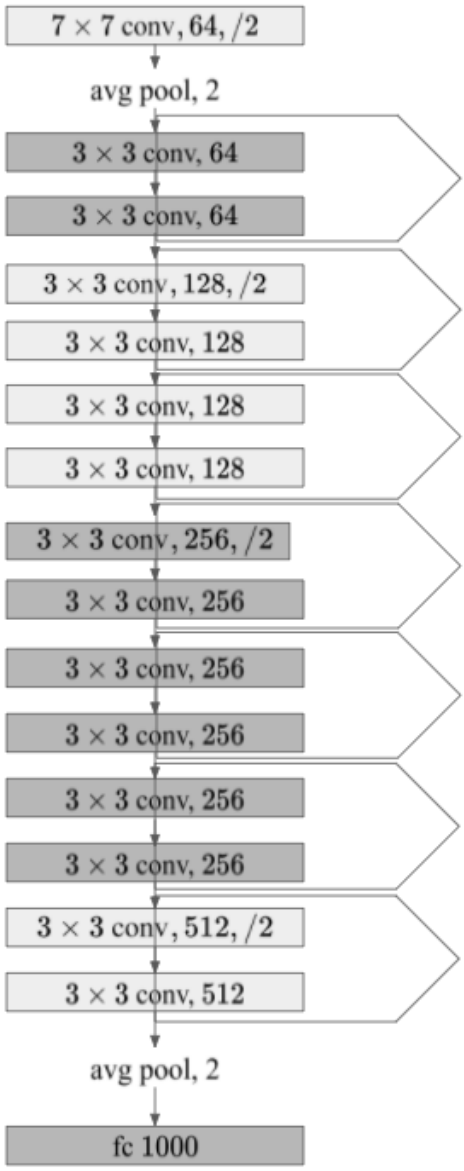



Figura 4.9: Diagrama de un ejemplo de ResNet. Imagen basada en [16].

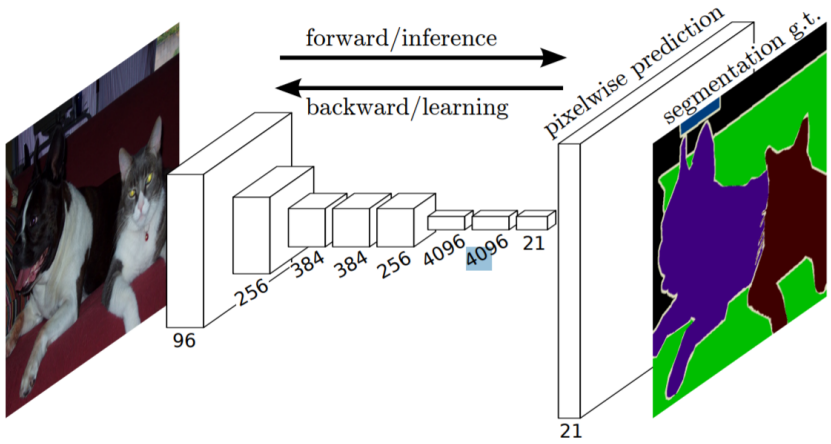



Figura 4.10: Red completamente convolucional (FCN). Imagen extraída de [40].

codificador-decodificador.

La U-Net juega un rol fundamental en el desarrollo de este trabajo. Esta, es una red neuronal convolucional especializada en segmentación de imágenes médicas. Esta arquitectura ganó el *Cell Tracking Challenge at ISBI 2015*. Como mencionamos, esta red es un ED con conexiones de salto, en el contexto específico de la U-Net, el codificador lleva el nombre de **camino contractivo** y al decodificador lo llamaremos **camino expansivo**. El camino contractivo consiste en varios bloques contractivos que aplican dos convoluciones de $3 \times 3$, una ReLu como activación y por último un max-pooling. El camino expansivo consiste en varios bloques expansivos que aplican dos convoluciones de $3 \times 3$ seguido de un sobremuestreo. Las conexiones de salto consisten que la entrada de cada bloque expansivo se forma concatenando la salida del bloque expansivo inmediato anterior con salida del bloque contractivo correspondiente (ver Figura 4.11). El artículo original de la U-Net no aplica relleno en ninguna capa convolucional, y no especifica que técnica de sobremuestreo utiliza. El no utilizar el relleno resulta que las dimensiones de la predicción inicialmente no coincidan con las de la verdad fundamental, así que en este trabajo, para evitar esa problemática, cualquier variante que se utilice de la U-Net será con convoluciones con relleno.

La última arquitectura que veremos es la **ResUnet**. Es natural querer implementar

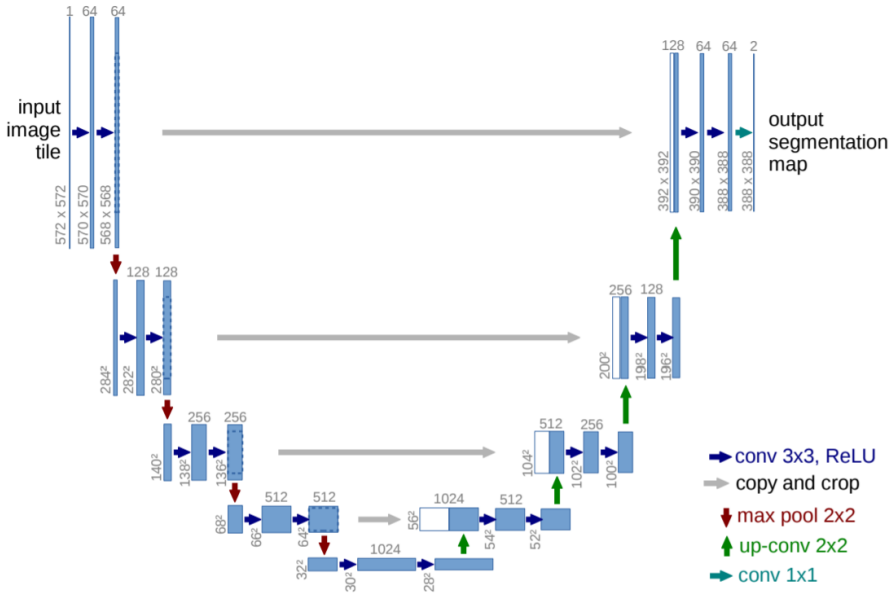



Figura 4.11: Arquitectura U-Net. Imagen extraída de [4].

las virtudes del aprendizaje residual en cualquier aplicación del aprendizaje profundo, en particular, en la U-Net para la segmentación. En [41] se propone justamente esto, una arquitectura inspirada en la U-Net y en la ResNet: la ResUnet. Debido a que las Neural ODEs pueden interpretarse como versiones continuas de la ResNet, es natural tomar una arquitectura con las características principales de la U-Net, y pasarla al paradigma de las Neural ODEs. Ésta arquitectura se puede visualizar en la Figura 4.12.

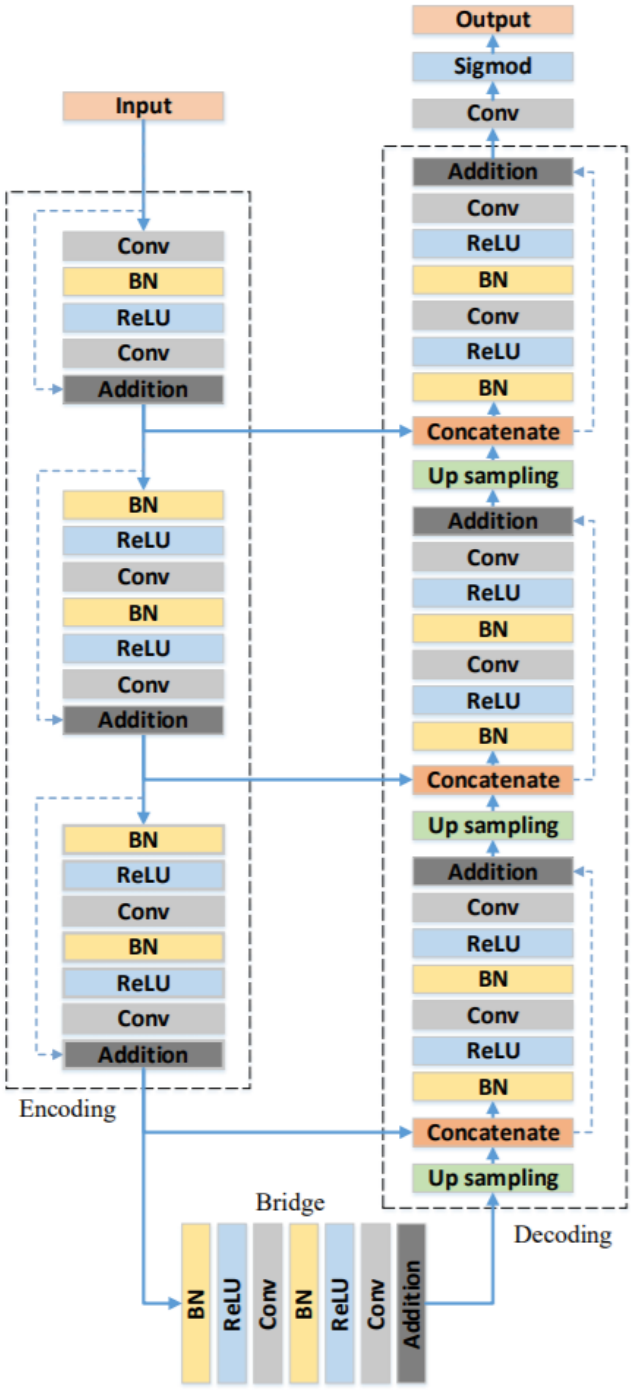



Figura 4.12: Diagrama de la ResUnet. Imagen extraída de [41].

# Capítulo 5

# Neural ODEs

Las **ecuaciones diferenciales ordinarias neuronales** (**Neural ODEs**) [2], son el modelo principal tratado en este trabajo.

Recordemos que la salida de una ResNet, donde la entrada es $\mathbf{x} \in \mathbb{R}^D$ está dada por

$$\widehat{\mathbf{y}} = \mathbf{z}_T, \quad \mathbf{z}_{k+1} = \mathbf{z}_k + f_k(\mathbf{z}_k; \boldsymbol{\theta}_k), \quad \mathbf{z}_0 = \mathbf{x}, \tag{5.1}$$

donde $f_k$ es el $k$-ésimo ResBlock y $T$ es la cantidad de ResBlocks.

**Nota 5.1.** *En las ResNet que vimos la entrada es un tensor* $\mathbf{x} \in \mathbb{R}^{C\times H\times W}$*, sin embargo para esta discusión no nos servirá esta estructura tensorial y podemos ver a* $\mathbf{x}$ *como un vector en* $\mathbb{R}^{CHW}$*.*

En particular, nos interesan las ResNet donde todos los ResBlocks son iguales; es decir, todos los $f_k$ son iguales.

$$\mathbf{z}_{k+1} = \mathbf{z}_k + f(\mathbf{z}_k; \boldsymbol{\theta}_k), \quad \mathbf{z}_0 = \mathbf{x}. \tag{5.2}$$

La idea de las Neural ODEs es considerar $\mathbf{z}$ como una función continua. La familia de modelos a la que le llamaremos Neural ODEs es la siguiente

$$\widehat{\mathbf{y}} = \mathbf{z}(T), \quad \dot{\mathbf{z}}(t) = f(\mathbf{z}(t); \boldsymbol{\theta}(t)), \quad \mathbf{z}(0) = \mathbf{x}, \quad [0, T] \tag{5.3}$$

En principio también es necesario considerar $\boldsymbol{\theta}$ una función continua, sin embargo, tiene la desventaja de que no es tan claro como se inicializaría ni como se actualizaría $\boldsymbol{\theta}$. En [42]

se propone una forma de hacer esto, sin embargo por simplicidad, consideremos que $\boldsymbol{\theta}$ es constante, *i.e.* $\boldsymbol{\theta} \in \mathbb{R}^{N_\theta}$ como se hace en [2]. En principio esto significaría que la ResNet tiene los mismos pesos en cada ResBlock.

Para observar mejor como están relacionadas las ecuaciones (5.1) y (5.3), veremos lo siguiente. Del primer teorema fundamental del cálculo, podemos reescribir (5.3) como

$$\mathbf{z}(T) = \mathbf{z}(0) + \int_0^T f(\mathbf{z}(\tau))\mathrm{d}\tau,$$

(Para simplificar la notación, estamos obviando que $f$ está parametrizada por $\boldsymbol{\theta}$). Mediante sumas de Riemann podemos dividir el segmento $[0, T]$ en $K$ partes iguales de longitud $h = 1/K$ y definimos

$$\widetilde{\mathbf{z}}_k = \mathbf{z}(0) + h\sum_{\kappa=0}^{k-1} f(\mathbf{z}(\kappa h)),$$

para $k = 0, ..., K$. De esta manera, tenemos que $\mathbf{z}(0) = \widetilde{\mathbf{z}}_0$ y $\mathbf{z}(kh) \approx \widetilde{\mathbf{z}}_k$, donde la aproximación se vuelve igualdad cuando $h \to 0$ y . Si definimos $\mathbf{z}_k = \mathbf{z}(hk)$, tenemos

$$\begin{aligned}
\widetilde{\mathbf{z}}_{k+1} &= \mathbf{z}_0 + h\sum_{\kappa=0}^{k} f(\mathbf{z}_\kappa) \\
&= \mathbf{z}_0 + h\sum_{\kappa=0}^{k-1} f(\mathbf{z}_\kappa) + hf(\mathbf{z}_k) \\
&= \widetilde{\mathbf{z}}_k + hf(\mathbf{z}_k) \\
&\approx \widetilde{\mathbf{z}}_k + hf(\widetilde{\mathbf{z}}_k).
\end{aligned}$$

De aquí surge la motivación del **método de Euler**, que justamente consiste en aproximar $\mathbf{z}(T)$, calculando iterativamente

$$\widetilde{\mathbf{z}}_{k+1} = \widetilde{\mathbf{z}}_k + hf(\widetilde{\mathbf{z}}_k), \quad \widetilde{\mathbf{z}}_0 = \mathbf{z}(0) = \mathbf{x},$$

escogiendo un $h$ pequeño. Notemos que si tomamos $h = 1$ obtenemos justamente (5.1). El que la ResNet tenga la forma del método de Euler para resolver ecuaciones diferenciales sirve como motivación para definir las Neural ODEs. Resolver (5.3) en la práctica se realiza numéricamente; son diversos los métodos numéricos que nos proporciona la teoría de ecuaciones diferenciales para resolver (5.3) (y sus variantes), entre otros tenemos: método del

punto medio, Runge-Kutta, entre otros. Sin embargo, para esta discusión, consideraremos simplemente el algoritmo de la ecuación diferencial como una caja negra. En la práctica además, siempre suele tomarse $T = 1$.

La función $f$ es conocida como el **flujo**, y en [2] se toma una variante de (5.3), donde consideran que el flujo puede depender también explícitamente del tiempo, en la práctica, esto se traduce a concatenar el tiempo al tensor de entrada $\mathbf{x}$ a la dimensión de los canales. En el caso de las convoluciones y en particular de la red que utilizaremos nosotros esto significa que las imágenes que originalmente son tensores $\mathbf{x} \in \mathbb{R}^{C\times H\times W}$, las extenderemos a $\mathbf{x}_t \in \mathbb{R}^{(C+1)\times H\times W}$. En este artículo también se considera $\boldsymbol{\theta}$ constante. Además, se puede agregar mayor generalidad si consideramos el intervalo de integración $[t_0, t_1]$.

$$\widehat{\mathbf{y}} = \mathbf{z}(t_1), \quad \dot{\mathbf{z}}(t) = f(\mathbf{z}(t), t; \boldsymbol{\theta}), \quad \mathbf{z}(t_0) = \mathbf{x}. \tag{5.4}$$

## 5.1. Actualización de pesos

El calcular la salida $\mathbf{y}$, sin importar el método que escojamos para resolver la ODE, es una secuencia de operaciones, sin embargo la enorme cantidad de operaciones que deben realizarse, hace que calcular el gradiente $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ resulte en una tarea muy costosa en caso de usarse el método tradicional de propagación hacia atrás. El método que proponen en [2] es el **método de la adjunta**. El método es el siguiente:

Resolvemos

$$\dot{\boldsymbol{\lambda}}^\top(t) = -\boldsymbol{\lambda}^\top(t)\frac{\partial f(\mathbf{z}(t), t; \boldsymbol{\theta})}{\partial \mathbf{z}(t)}, \quad \boldsymbol{\lambda}^\top(t_1) = \frac{\partial L}{\partial \mathbf{z}(t_1)}, \tag{5.5}$$

atrás en el tiempo en el intervalo $[t_0, t_1]$. Con esto se puede calcular

$$\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\theta}} = -\int_{t_1}^{t_0} \boldsymbol{\lambda}^\top(t)\frac{\partial f(\mathbf{z}(t), t; \boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\mathrm{d}t, \tag{5.6}$$

que es lo que buscamos para poder aplicar un optimizador como el descenso del gradiente. Antes de dar la demostración de que el método funciona, mostraremos un ejemplo extraído de [43].

**Ejemplo 5.2.** *Consideremos la propagación hacia adelante con entrada $x \in \mathbb{R}$ está dada por la ecuación diferencial*

$$\widehat{y} = z(t), \quad \dot{z}(t) = wz(t), \quad z(t_0) = x$$

*donde $z : \mathbb{R} \to \mathbb{R}$, $w \in \mathbb{R}$, y*

$$L(z(t_1), y) = \frac{1}{2}(z(t_1) - y)^2.$$

*Queremos encontrar*

$$\frac{\partial L(z(t_1; w), y)}{\partial w}.$$

*La ecuación diferencial se puede resolver analíticamente, y encontramos que*

$$z(t) = e^{w(t-t_0)}x.$$

*A continuación resolvemos* (5.5)*, que en nuestro ejemplo se escribe*

$$\dot{\lambda}(t) = -\lambda \frac{\partial wz(t)}{\partial z(t)}, \quad \lambda(t_1) = \frac{\partial L}{\partial z(t_1)},$$

*y obtenemos*

$$\begin{aligned}
\lambda(t) &= \lambda(t_1)e^{-w(t-t_1)} \\
&= \frac{\partial L}{\partial z(t_1)}e^{-w(t-t_1)} \\
&= (z(t_1) - y)e^{-w(t-t_1)};
\end{aligned}$$

*el signo menos surge de que la ecuación diferencial se resuelve hacia atrás en el tiempo. Por último, resolvemos* (5.6)*:*

$$\begin{aligned}
\frac{\mathrm{d}L}{\mathrm{d}w} &= -\int_{t_1}^{t_0} \lambda(t)\frac{\partial wz(t)}{\mathrm{d}w}\mathrm{d}t \\
&= -\int_{t_1}^{t_0} (z(t_1) - y)e^{-w(t-t_1)}z(t)\mathrm{d}t \\
&= -\int_{t_1}^{t_0} (z(t_1) - y)e^{-w(t-t_1)}e^{w(t-t_0)}x\mathrm{d}t \\
&= (z(t_1) - y)xe^{w(t_1-t_0)}\int_{t_0}^{t_1} \mathrm{d}t \\
&= (z(t_1) - y)xe^{w(t_1-t_0)}(t_1 - t_0).
\end{aligned}$$

*En este ejemplo pequeño, podemos calcular el gradiente analíticamente para verificar que coinciden*

$$\begin{aligned}\frac{\mathrm{d}L(z(t_1))}{\mathrm{d}w} &= \frac{\mathrm{d}L}{\mathrm{d}z(t_1)}\frac{\mathrm{d}z(t_1)}{\mathrm{d}w} \\ &= \left(\frac{1}{2}\frac{\mathrm{d}(z(t_1)-y)^2}{\mathrm{d}z(t_1)}\right)\left(x\frac{\mathrm{d}e^{w(t_1-t_0)}}{\mathrm{d}w}\right) \\ &= (z(t_1)-y)xe^{w(t_1-t_0)}(t_1-t_0).\end{aligned}$$

A continuación mostraremos una demostración del método de la adjunta dada en [44].

**Teorema 5.3.** *Sea* $\mathbf{z}$ *una función que sigue la ODE* (5.4)*,* $L:\mathbb{R}^{N_z}\to\mathbb{R}$ *y* $\boldsymbol{\lambda}$ *definida por* (5.5)*. Entonces el gradiente con respecto a* $\boldsymbol{\theta}$ *está dado por* (5.6)*.*

*Demostración.* Definimos

$$F(\dot{\mathbf{z}}(t),\mathbf{z}(t),\boldsymbol{\theta},t) = \dot{\mathbf{z}}(t) - f(\mathbf{z}(t),\boldsymbol{\theta},t)$$

y

$$\psi = L(\mathbf{z}(t_1)) - \int_{t_0}^{t_1}\boldsymbol{\lambda}^\top(t)F(\dot{\mathbf{z}}(t),\mathbf{z}(t),\boldsymbol{\theta},t)\mathrm{d}t. \tag{5.7}$$

Para aligerar la notación, escribiremos $f \equiv f(\mathbf{z}(t),\boldsymbol{\theta},t)$ y $F \equiv F(\dot{\mathbf{z}}(t),\mathbf{z}(t),\boldsymbol{\theta},t)$. Ahora, tenemos que como $F=0$ por (5.4), así que

$$\frac{\mathrm{d}\psi}{\mathrm{d}\boldsymbol{\theta}} = \frac{\mathrm{d}L(\mathbf{z}(t_1))}{\mathrm{d}\boldsymbol{\theta}}.$$

Analizando la integral en (5.7)

$$\begin{aligned}\int_{t_0}^{t_1}\boldsymbol{\lambda}^\top(t)F\mathrm{d}t &= \int_{t_0}^{t_1}\boldsymbol{\lambda}^\top(t)(\dot{\mathbf{z}}(t)-f)\mathrm{d}t \\ &= \int_{t_0}^{t_1}\boldsymbol{\lambda}(t)\dot{\mathbf{z}}(t)\mathrm{d}t - \int_{t_0}^{t_1}\boldsymbol{\lambda}^\top(t)f\mathrm{d}t \end{aligned}\tag{5.8}$$

Integrando por partes el primer sumando de (5.8), vemos que

$$\begin{aligned}\int_{t_0}^{t_1}\boldsymbol{\lambda}^\top(t)\dot{\mathbf{z}}(t)\mathrm{d}t &= \boldsymbol{\lambda}^\top(t)\mathbf{z}(t)\big|_{t_0}^{t_1} - \int_{t_0}^{t_1}\mathbf{z}^\top(t)\dot{\boldsymbol{\lambda}}(t)\mathrm{d}t \\ &= \boldsymbol{\lambda}^\top(t_1)\mathbf{z}(t_1) - \boldsymbol{\lambda}^\top(t_0)\mathbf{x} - \int_{t_0}^{t_1}\dot{\boldsymbol{\lambda}}^\top(t)\mathbf{z}(t)\mathrm{d}t. \end{aligned}\tag{5.9}$$

Sustituyendo (5.9) en (5.8), obtenemos

$$\int_{t_0}^{t_1} \boldsymbol{\lambda}^\top(t) F \mathrm{d}t = \boldsymbol{\lambda}^\top(t_1)\mathbf{z}(t_1) - \boldsymbol{\lambda}(t_0)\mathbf{x} - \int_{t_0}^{t_1} \left(\dot{\boldsymbol{\lambda}}^\top(t)\mathbf{z}(t) + \boldsymbol{\lambda}^\top(t) f\right) \mathrm{d}t \tag{5.10}$$

Tomando la derivada con respecto a $\boldsymbol{\theta}$ en (5.10), nos queda

$$\frac{\mathrm{d}}{\mathrm{d}\boldsymbol{\theta}}\left(\int_{t_0}^{t_1} \boldsymbol{\lambda}^\top(t) F \mathrm{d}t\right) = \boldsymbol{\lambda}^\top(t_1)\frac{\mathrm{d}\mathbf{z}(t_1)}{\mathrm{d}\boldsymbol{\theta}} - \int_{t_0}^{t_1} \left(\dot{\boldsymbol{\lambda}}^\top(t)\frac{\mathrm{d}\mathbf{z}(t)}{\mathrm{d}\boldsymbol{\theta}} + \boldsymbol{\lambda}^\top(t)\frac{\mathrm{d}f}{\mathrm{d}\boldsymbol{\theta}}\right) \mathrm{d}t \tag{5.11}$$

Por la regla de la cadena, sabemos que

$$\frac{\mathrm{d}f}{\mathrm{d}\boldsymbol{\theta}} = \frac{\partial f}{\partial \boldsymbol{\theta}} + \frac{\partial f}{\partial \mathbf{z}(t)}\frac{\mathrm{d}\mathbf{z}(t)}{\mathrm{d}\boldsymbol{\theta}} \tag{5.12}$$

Sustituyendo (5.12) en (5.11), tenemos

$$\frac{\mathrm{d}}{\mathrm{d}\boldsymbol{\theta}}\left(\int_{t_0}^{t_1} \boldsymbol{\lambda}^\top(t) F \mathrm{d}t\right) = \boldsymbol{\lambda}^\top(t_1)\frac{\mathrm{d}\mathbf{z}(t_1)}{\mathrm{d}\boldsymbol{\theta}} - \int_{t_0}^{t_1} \left(\dot{\boldsymbol{\lambda}}^\top(t)\frac{\mathrm{d}\mathbf{z}(t)}{\mathrm{d}\boldsymbol{\theta}} + \boldsymbol{\lambda}^\top(t)\frac{\partial f}{\partial \boldsymbol{\theta}} + \boldsymbol{\lambda}^\top(t)\frac{\mathrm{d}f}{\mathrm{d}\mathbf{z}(t)}\frac{\mathrm{d}\mathbf{z}(t)}{\mathrm{d}\boldsymbol{\theta}}\right) \mathrm{d}t. \tag{5.13}$$

Sustituyendo nuestra hipótesis (5.5) en (5.13), tenemos

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}\boldsymbol{\theta}}\left(\int_{t_0}^{t_1} \boldsymbol{\lambda}^\top(t) F \mathrm{d}t\right) &= \frac{\partial L}{\partial \mathbf{z}(t_1)}\frac{\mathrm{d}\mathbf{z}(t_1)}{\mathrm{d}\boldsymbol{\theta}} - \int_{t_0}^{t_1} \left(\dot{\boldsymbol{\lambda}}^\top(t)\frac{\mathrm{d}\mathbf{z}(t)}{\mathrm{d}\boldsymbol{\theta}} + \boldsymbol{\lambda}^\top(t)\frac{\partial f}{\partial \boldsymbol{\theta}} - \dot{\boldsymbol{\lambda}}^\top(t)\frac{\mathrm{d}\mathbf{z}(t)}{\mathrm{d}\boldsymbol{\theta}}\right) \mathrm{d}t \\
&= \frac{\partial L}{\partial \mathbf{z}(t_1)}\frac{\mathrm{d}\mathbf{z}(t_1)}{\mathrm{d}\boldsymbol{\theta}} - \int_{t_0}^{t_1} \boldsymbol{\lambda}^\top(t)\frac{\partial f}{\partial \boldsymbol{\theta}} \mathrm{d}t.
\end{aligned} \tag{5.14}$$

Finalmente, calculando el gradiente que nos interesa, aplicando la regla de la cadena de nuevo tenemos

$$\begin{aligned}
\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\theta}} &= \frac{\mathrm{d}\psi}{\mathrm{d}\boldsymbol{\theta}} \\
&= \frac{\mathrm{d}L}{\mathrm{d}\mathbf{z}(t_1)}\frac{\mathrm{d}\mathbf{z}(t_1)}{\mathrm{d}\boldsymbol{\theta}} - \frac{\mathrm{d}}{\mathrm{d}\boldsymbol{\theta}}\left(\int_{t_0}^{t_1} \boldsymbol{\lambda}^\top(t) F \mathrm{d}t\right) \\
&= \frac{\mathrm{d}L}{\mathrm{d}\mathbf{z}(t_1)}\frac{\mathrm{d}\mathbf{z}(t_1)}{\mathrm{d}\boldsymbol{\theta}} - \frac{\partial L}{\partial \mathbf{z}(t_1)}\frac{\mathrm{d}\mathbf{z}(t_1)}{\mathrm{d}\boldsymbol{\theta}} + \int_{t_0}^{t_1} \boldsymbol{\lambda}^\top(t)\frac{\partial f}{\partial \boldsymbol{\theta}} \mathrm{d}t \\
&= \int_{t_0}^{t_1} \boldsymbol{\lambda}^\top(t)\frac{\partial f}{\partial \boldsymbol{\theta}} \mathrm{d}t \\
&= -\int_{t_1}^{t_0} \boldsymbol{\lambda}^\top(t)\frac{\partial f}{\partial \boldsymbol{\theta}} \mathrm{d}t.
\end{aligned}$$

□

La intuición detrás del método, es que escoger $\boldsymbol{\lambda}$ de esta manera, nos evita tener que calcular expresiones que serían muy difíciles de obtener, específicamente $\frac{\mathrm{d}\mathbf{z}(t)}{\mathrm{d}\boldsymbol{\theta}}$ para un tiempo arbitrario.

**Nota 5.4.** *La derivada total $\frac{\mathrm{d}f}{\mathrm{d}\boldsymbol{\theta}}$ no debe confundirse con la derivada parcial $\frac{\partial f}{\mathrm{d}\boldsymbol{\theta}}$ en* (5.12)*. Por ejemplo, si tenemos $f(z(t), w, t) = wz(t)$, y tenemos que $z(t)$ depende de $w$, la derivada parcial $\frac{\partial f}{\partial w}$, no considera los cambios de $z(t)$ con respecto a $w$, de manera que tendríamos $\frac{\partial f}{\partial w} = z(t)$. Mientras que la derivada total sí contempla los cambios de $z(t)$ con respecto a $\theta$, teniendo entonces $\frac{\mathrm{d}f}{\mathrm{d}w} = \frac{\partial f}{\partial w} + \frac{\partial f}{\partial z(t)}\frac{\mathrm{d}f}{\mathrm{d}w}$.*

Una vez hallado el gradiente $\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\theta}}$, procedemos a optimizar los parámetros como en cualquier método tradicional. Además de $\boldsymbol{\theta}$, uno puede considerar los puntos inicial y de integración $t_0$ y $t_1$ como variables a optimizar, dándole algo más de libertad al modelo. En estos casos, es posible encontrar los gradientes $\frac{\mathrm{d}L}{\mathrm{d}t_0}$ y $\frac{\mathrm{d}L}{\mathrm{d}t_1}$ y optimizarlos junto con el resto de parámetros.

Para resumir, el Algoritmo 2 [2] muestra el entrenamiento de una Neural ODE utilizando el descenso de gradiente para optimizar sus parámetros, que a su vez, utiliza la subrutina 1 para calcular los gradientes.

**Entrada:** parámetros $\boldsymbol{\theta}$, tiempo inicial $t_0$, tiempo final $t_1$, estado final $\mathbf{z}(t_1)$, gradiente del estado final $\frac{\partial L}{\partial \mathbf{z}(t_1)}$.
**Salida :** gradiente $\frac{\mathrm{d}L}{\mathrm{d}\theta}$.
Resolver la ecuación diferencial con punto inicial $\boldsymbol{\lambda}^\top(t) = -\boldsymbol{\lambda}^\top(t)\frac{\partial f(\mathbf{z}(t), t\boldsymbol{\theta})}{\partial \mathbf{z}(t)}$, $\lambda(t_1) = \frac{\partial L}{\partial \mathbf{z}(t_1)}$ en el intervalo $[t_1, t_0]$.
Resolver la integral $\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\theta}} = -\int_{t_1}^{t_0} \boldsymbol{\lambda}^\top(t)\frac{\partial f(\mathbf{z}(t), t; \boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\mathrm{d}t$.
**devolver** $\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\theta}}$.

**Algoritmo 1:** Cálculo del gradiente.

El usar Neural ODEs, tiene algunas propiedades interesantes respecto a otras redes convolucionales más tradicionales; destacamos dos en particular:

**Entrada:** dato $\mathbf{x}$, su etiqueta $y$, los parámetros del modelo $\boldsymbol{\theta}$, el tiempo inicial $t_0$ y el tiempo final $t_1$.

**Salida :** parámetros optimizados $\boldsymbol{\theta}^*$.

$\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}$.

**mientras** $||\boldsymbol{\theta}^* - \boldsymbol{\theta}|| \leq \epsilon$ **hacer**

- $\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}$.
- Calculamos $\mathbf{z}(t_1)$ resolviendo $\dot{\mathbf{z}}(t) = f(\mathbf{z}(t), t; \boldsymbol{\theta}), \quad \mathbf{z}(t_0) = \mathbf{x}$.
- $\widehat{y} \leftarrow \mathbf{z}(t_1)$.
- Calculamos $L(\widehat{y}, y)$.
- Calculamos $\frac{\partial L}{\partial \mathbf{z}(t_1)}$.
- Calculamos $\frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\theta}}$ usando el Algoritmo 1, pasándole como entrada $\boldsymbol{\theta}$, $t_0$, $t_1$, $\mathbf{z}(t_1)$, $\frac{\partial L}{\partial \mathbf{z}(t_1)}$.
- $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \frac{\mathrm{d}L}{\mathrm{d}\boldsymbol{\theta}}$.

**fin**

$\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}$. **devolver** $\boldsymbol{\theta}^*$.

**Algoritmo 2:** Descenso de gradiente en una Neural ODE.

1. El cálculo del gradiente, tiene complejidad espacial $O(1)$ con respecto a la profundidad de la red. Esto se debe a que no es necesario guardar los valores intermedios de los gradientes para realizar la propagación hacia atrás.

2. Es posible intercambiar precisión por tiempo según sea conveniente. Esto se debe a que la mayoría de los métodos numéricos modernos (*e.g.* Runge-Kutta) poseen parámetros que moderan el error numérico.

# Capítulo 6

# Experimentos y Resultados

En este capítulo, se presentarán los experimentos realizados, las métricas consideradas y los resultados obtenidos. Este trabajo, continua desarrollando la línea de segmentar el parásito *T. cruzi*. Cabe mencionar que en cierta medida, esta tesis es una continuación de [23], tesis de maestría presentada por Ojeda A., en el que como se mencionó en el Capítulo 1, segmentó al parásito *T. cruzi* mediante una modificación de la ResNet.

## 6.1. El conjunto de datos

El conjunto de datos utilizado consiste en imágenes de muestras de sangre de $256 \times 256$ pixeles que contienen de cero a dos parásitos *T.cruzi*, en particular, 62 imágenes no contienen parásitos y el 978 restante contiene al menos un parásito, obteniendo así un total de 1040 imágenes. Éstas imágenes, se dividieron en conjuntos de tamaños 626, 207 y 207 que respectivamente, que corresponden a los conjuntos de entrenamiento, prueba y validación. Además, se realizaron traslaciones aleatorias, incrementando el tamaño del conjunto de entrenamiento a 3824 imágenes. Cabe mencionar, que las imágenes fueron tomadas utilizando un microscopio y teñidas para visualizar con mayor facilidad la morfología de los parásitos. Por último, se hace un agradecimiento especial a Ojeda A., quien, etiquetó el conjunto segmentando manualmente los pixeles pertenecientes al fondo de los pertenecien-

tes a la clase principal.

Resumiendo, el conjunto de datos, cuenta con 626 imágenes en formato RGB (tensores en $\mathbb{R}^{3\times256,256}$) con sus respectivas etiquetas, las cuales son imágenes binarias (tensores en $\{0,1\}^{256\times256}$), en donde los píxeles con valor 1 denotan que en esa posición está presente parte del parásito.

## 6.2. Modelos

Se experimentó con tres modelos diferentes: la U-Net [4] de la cual se habló en el Capítulo 3.1, también, con una arquitectura que combina la U-Net con el paradigma de la ResNet a la cual se le denominó ResUNet y que se propone en [15], y por último la UNODE que se propone también en [15].

### 6.2.1. U-Net

La U-Net con la cual se experimentó en este trabajo, consiste en cuatro bloques codificadores y cuatro bloques decodificadores. Los bloques codificadores están compuestos de la siguiente manera: convolución con kernel de tamaño 3 y zero padding, seguido de un max pooling con kernel de tamaño 2, cada convolución duplica la cantidad de canales. Por otro lado, los bloques decodificadores son de la siguiente forma: convolución con kenrel de tamaño 3 y zero padding, seguido de una convolución transpuesta con kernel de tamaño 2, donde cada convolución divide a la mitad el número de canales. Además, la entrada del $k$-ésimo bloque codificador es concatenada con la salida del $4-k$-ésimo bloque decodificador y se toma como la entrada del $(4-k)+1$-ésimo bloque decodificador. La estructura de esta red se puede ver en la Figura 6.1.

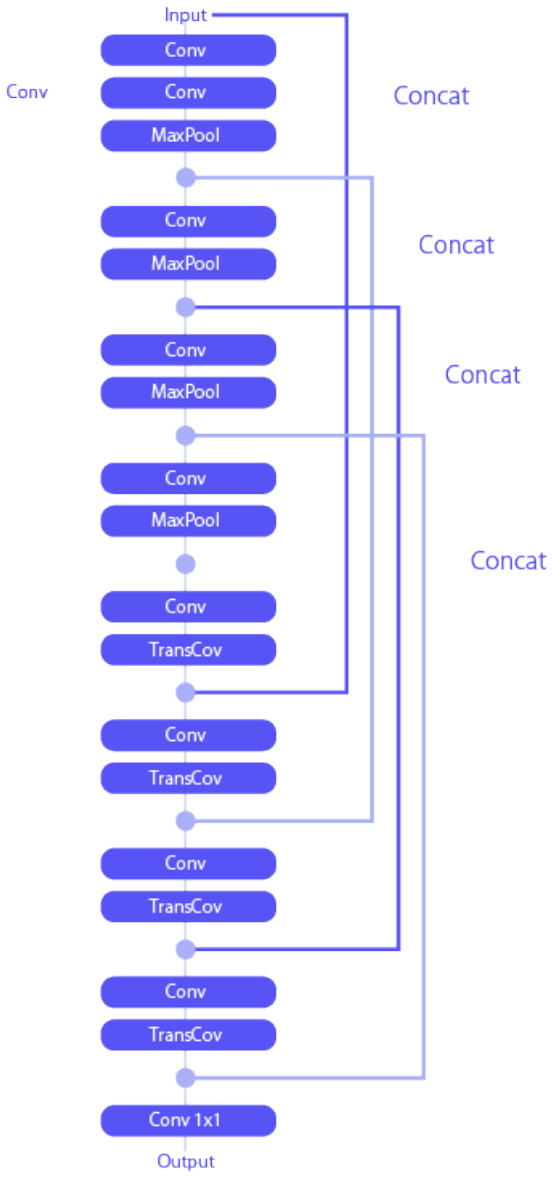



Figura 6.1: Arquitectura U-Net utilizada en los experimentos.

### 6.2.2. Flujo

En el Capítulo 5, se definió el flujo para una Neural ODE. Recordando, una Neural ODE sigue la ecuación

$$\dot{\mathbf{z}}(t) = f(\mathbf{z}(t); \boldsymbol{\theta}), \quad \mathbf{z}(0) = \mathbf{x},$$

con $t \in [0, 1]$, y la función $f$ es a la que se define como flujo.

Ahora, ya que las ResNets pueden ser vistas como una una discretización de las Neural ODEs, tiene sentido definir el flujo para las ResNet. Nuevamente en el Capítulo 5, se vio que la ResNet sigue la siguiente fórmula

$$\mathbf{z}_{k+1} = \mathbf{z}_k + f(\mathbf{z}_k; \boldsymbol{\theta}), \quad \mathbf{z}_0 = \mathbf{x},$$

para $k = 0, ..., K-1$. En el contexto de las ResNet, a $f$ se les llamó bloques ResNet. Para unificar conceptos, a partir de ahora se le llamará flujo en ambos contextos, y los ResBlock serán la función $\widetilde{f} : \mathbf{x} \mapsto \mathbf{x} + f(\mathbf{x})$.

En este trabajo, el flujo que se utilizó se puede ver en la Figura 6.2:

### 6.2.3. ResUNet

La ResUNet que se evaluó, es similar a la U-Net, en el sentido de que también cuenta con cuatro bloques codificadores y cuatro bloques decodificadores. Al igual que en la U-Net, se realizan concatenaciones de bloques en el camino decodificador con el bloque codificador, como conexiones de salto. Lo único que cambia es la forma de cada bloque codificador y decodificador, este está compuesto como se observa en la Figura 6.3. Además, en la Figura 6.4 se puede visualizar un bloque de una Neural ODE.

### 6.2.4. UNODE

La UNODE comparte la mayoría de los elementos de la ResUNet, con la diferencia de que los bloques ResNet son Neural ODEs. Es decir que cada bloque va a resolver una ecuación diferencial, en todas se usa el mismo flujo $f$, con sus propios parámetros. Supóngase que se está entrenando el bloque $B$, se tendría, entonces, la ecuación diferencial

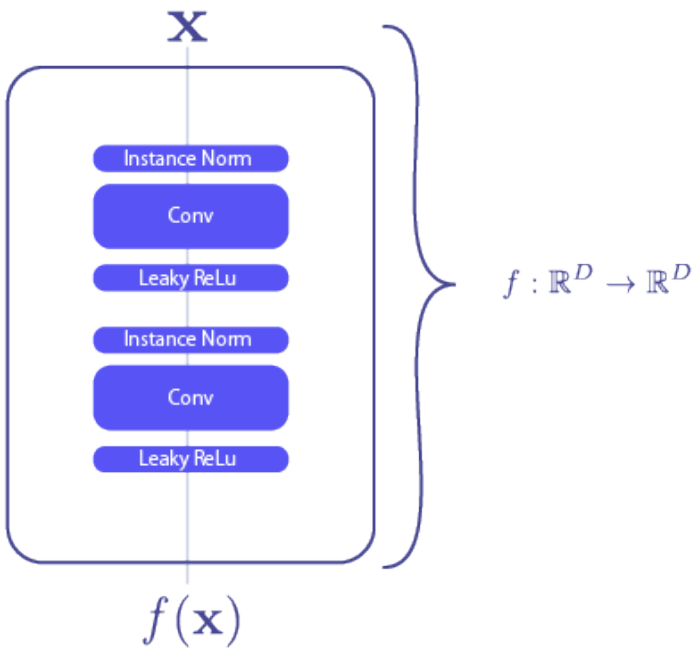



Figura 6.2: Flujo que se utilizará en la ResUNet y en la UNODE.

$$\dot{\mathbf{z}}(t) = f(\mathbf{z}(t); \boldsymbol{\theta}_B), \quad \mathbf{z}(0) = \mathbf{x}_B, \quad \mathbf{y}_B = \mathbf{z}(1),$$

donde $\mathbf{x}_B$ y $\mathbf{y}_B$ son la entrada y la salida del bloque $B$ respectivamente. La arquitectura de la UNODE se pude observar en la Figura 6.4, donde cada bloque es una neural ODE, que se representa en la Figura 6.5, y en la Figura 6.2 se puede ver el flujo $f$.

## 6.3. Métricas

En el Capítulo 3.5.2 se mostraron algunas métricas comunes en la segmentación. De las métricas consideradas en ese capítulo, se evaluarán en los resultados Precision, Recall, Accuracy, Dice Coefficient (F1-score) y Intersection over Union (IoU).

Existe un problema a la hora de utilizar estas métricas en segmentación: supóngase que la imagen consiste únicamente en píxeles que corresponden al fondo (0). Este caso ocurre si y sólo si TP = 0 y FN = 0, con lo que

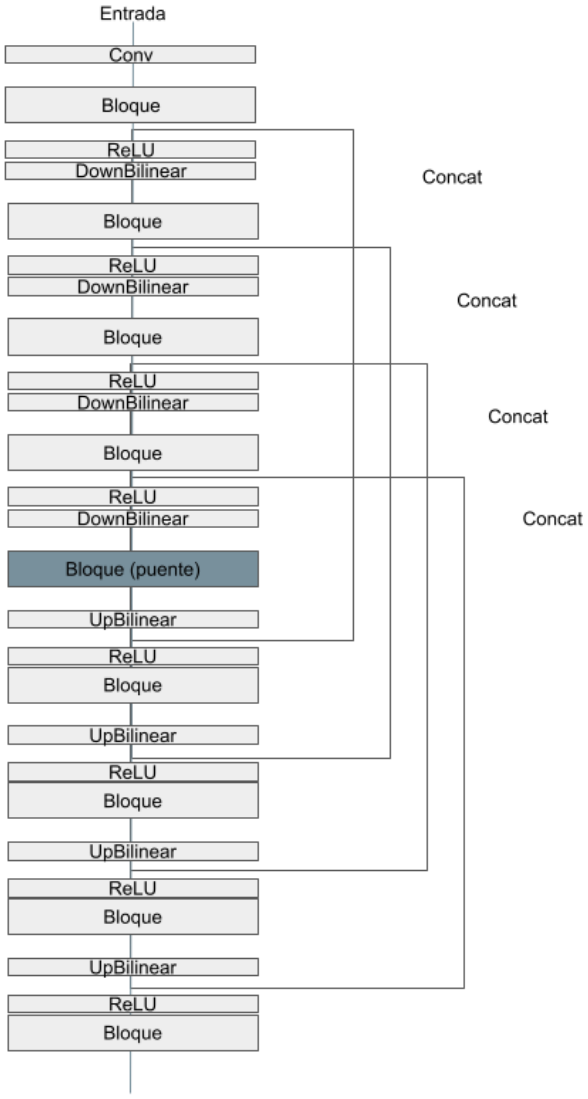



Figura 6.3: Arquitectura de la ResUNet y de la UNODE. En el caso de la ResUNet, se sustituye el "Bloque" por el que se presenta en la Figura 6.4, y para la UNODE se toma el que aparece en la Figura 6.5.

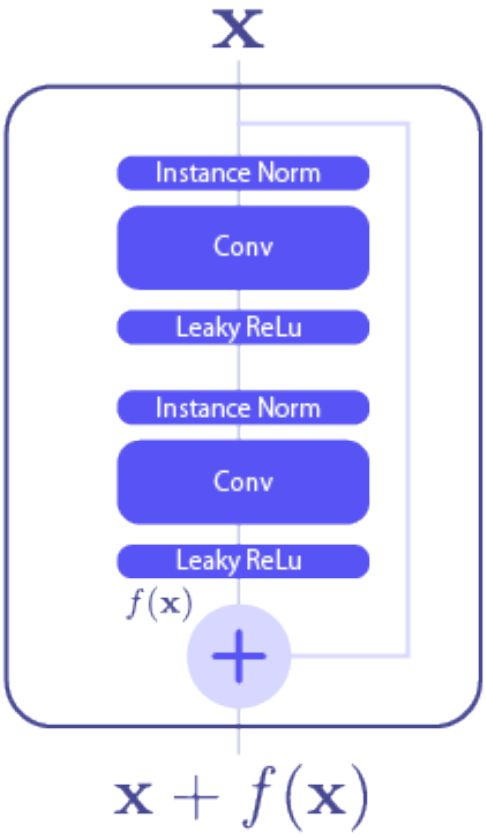



Figura 6.4: Resblock $\tilde{f}$.

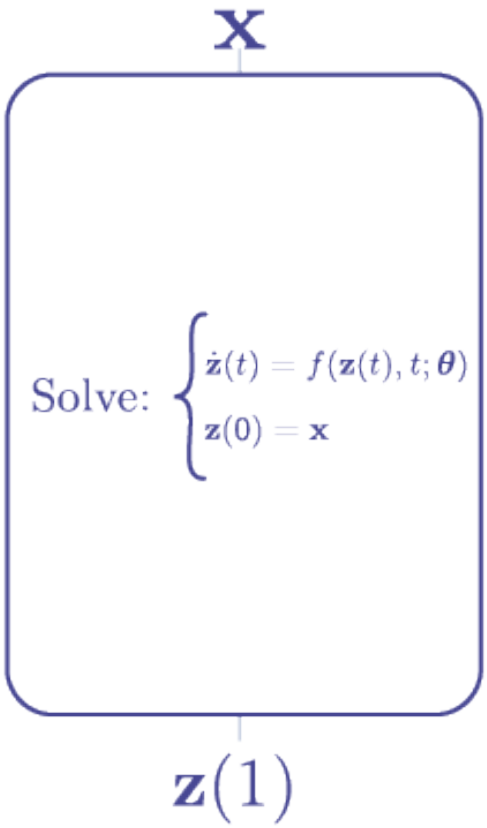



Figura 6.5: Bloque de la UNODE.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} = \frac{0}{0},$$
$$\text{IoU} = \frac{\text{TP}}{\text{TP} + \text{FN} + \text{FP}} = \frac{0}{\text{FP}},$$
$$\text{F1} = 2\frac{0}{2\text{TP} + \text{FN} + \text{FP}} = \frac{0}{\text{FP}}.$$

Esto es problemático, porque no importa lo bien que esté hecha la segmentación, siempre se obtendrá 0 en estas métricas (y el Recall nunca está bien definido), lo cual no parece obedecer a la intuición. Aún peor, si la segmentación se realiza perfectamente ninguno de estas métricas está definida. Al conjunto de imágenes en las que TP = FN = 0, se le llamará los casos especiales.

Debido al inconveniente que presentan los casos especiales, en este trabajo se evaluarán las métricas de tres formas diferentes:

1. En la primera forma, se considerará que cuando se de un caso especial, las métricas que no estén definidas sean iguales a 0.

2. En la segunda forma, se evaluarán las métricas en el conjunto de imágenes que no son un caso especial.

3. En la tercera forma, se considerará cada píxel individualmente. Normalmente se calcula para cada imagen $\mathbf{x}^{(m)}$ su respectiva matriz de confusión $\mathbf{C}_m$, y con esta matriz de confusión se obtiene la métrica $\rho(\mathbf{C}_m)$. Ahora lo que se hará, será calcular una matriz de confusión $\mathbf{C}$ para todo el conjunto $\mathbf{X}$, y con ello se calculará la métrica $\rho(\mathbf{C})$. De hecho,
$$\mathbf{C} = \sum_m \mathbf{C}_m.$$

En particular esta última opción debería cambiar continuamente conforme la segmentación va mejorando, lo cual es deseable en una métrica. A estas, se le llamarán: forma 1, 2 y 3 respectivamente.

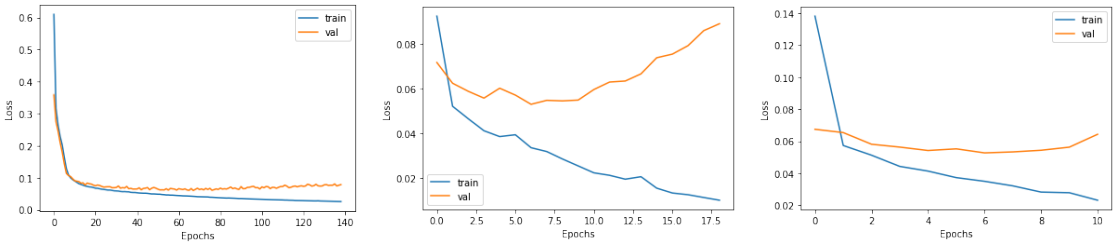
## 6.4. Experimentos

### 6.4.1. Generalidades

Las características que se comparten a lo largo de todos los experimentos son: tamaño de lote de 8, optimizador ADAM con una razón de aprendizaje de $10^{-4}$ y función de pérdida entropía cruzada binaria.

### 6.4.2. Diferentes tolerancias

Como se discutió en el Capítulo 5, teóricamente, es posible intercambiar velocidad por precisión en las Neural ODEs, ajustando la tolerancia numérica en el método utilizado para resolver la ODE asociada. Se realizaron experimentos con tres diferentes tolerancias, las cuales son: $10^{-2}$, $10^{-3}$ y $10^{-4}$. En la Figura 6.6 se observan las gráficas de sus respectivas funciones de pérdida. Además, en las Tablas 6.1a, 6.1b, 6.1c se muestran los resultados obtenidos usando las métricas descritas anteriormente. Adicionalmente, se midió el tiempo promedio de entrenamiento por lote de la UNODE con cada una de las anteriores tolerancias, con tolerancia de $10^{-5}$ añadido a las tolerancias anteriormente discutidas. Se puede ver una comparativa de tiempos por iteración promedio en la Figura 6.7. Por último, en la Figura 6.8, se puede visualizar las diferencias entre las segmentaciones obtenidas por la UNODE con diferentes tolerancias en una muestra de cinco imágenes, en las que, tres imágenes contienen al menos un parásito y dos de ellas no contienen ningún parásito.




(a) Tolerancia $10^{-2}$ (b) Tolerancia $10^{-3}$ (c) Tolerancia $10^{-4}$

Figura 6.6: Comparación de funciones de pérdida para distintos niveles de tolerancias.

| **Forma 1** | $10^{-2}$ | $10^{-3}$ | $10^{-4}$ |
|---|---|---|---|
| Accuracy | 0.976 | **0.980** | 0.979 |
| Dice | 0.768 | **0.789** | 0.780 |
| IoU | 0.654 | **0.684** | 0.671 |
| Precision | 0.745 | **0.766** | 0.753 |
| Recall | 0.808 | **0.828** | 0.823 |

(a) Forma 1

| **Forma 2** | $10^{-2}$ | $10^{-3}$ | $10^{-4}$ |
|---|---|---|---|
| Accuracy | 0.978 | **0.980** | 0.979 |
| Dice | 0.815 | **0.838** | 0.828 |
| IoU | 0.695 | **0.727** | 0.713 |
| Precision | 0.791 | **0.813** | 0.800 |
| Recall | 0.857 | **0.879** | 0.873 |

(b) Forma 2

| **Forma 3** | $10^{-2}$ | $10^{-3}$ | $10^{-4}$ |
|---|---|---|---|
| Accuracy | 0.977 | **0.981** | 0.979 |
| Dice | 0.800 | **0.830** | 0.819 |
| IoU | 0.666 | **0.710** | 0.693 |
| Precision | 0.756 | **0.792** | 0.777 |
| Recall | 0.848 | **0.872** | 0.865 |

(c) Forma 3

Tabla 6.1: Comparación de la UNODE con distintas tolerancias.

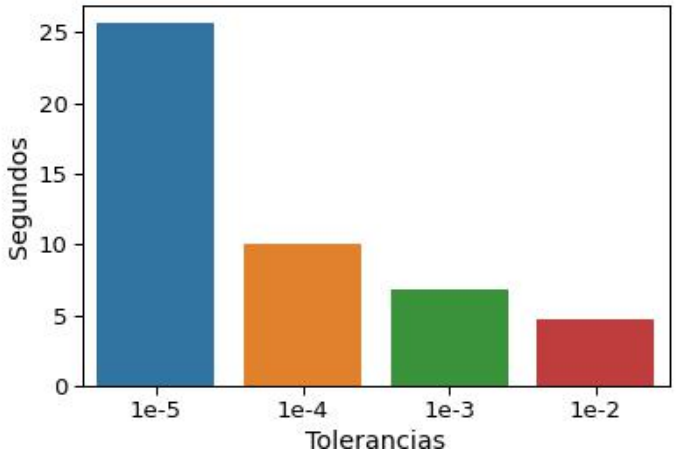
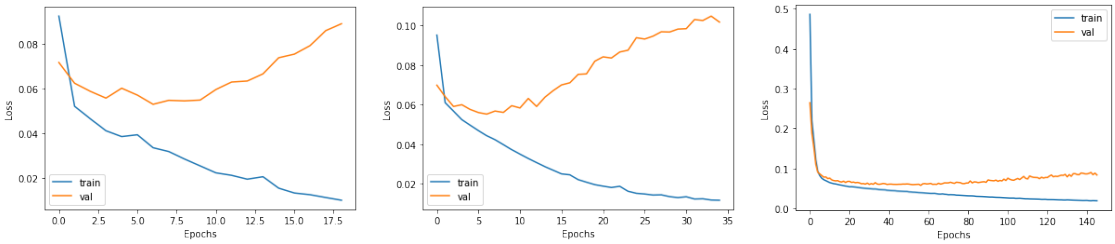



Figura 6.7: Comparativa de tiempos de la UNODE con diferentes tolerancias.


Figura 6.8: Segmentaciones realizadas por la UNODE con diferentes tolerancias.




(a) UNODE (b) ResUNet (c) U-Net

Figura 6.9: Comparación de funciones de pérdida para distintas arquitecturas.

### 6.4.3. Comparación con U-Net y ResUNet

En las Tablas 6.2a, 6.2b, 6.2c, se muestran las métricas para cada uno de los modelos; aquí, se está considerando la UNODE con tolerancia $10^{-3}$. Además, se presenta una comparativa entre las segmentaciones realizadas por las diferentes arquitecturas en la Figura 6.10. Las funciones de pérdida de cada modelo se muestran en la Figura 6.9.

### 6.4.4. Aumento de datos y abandono

Al observar las curvas de entrenamiento, se notaron indicadores que podrían indicar sobreajuste, pues hay una cantidad considerable de épocas. El conjunto de datos ya fue aumentado vía traslaciones, sin embargo se experimentó con otras transformaciones, como rotaciones, acercamientos y reflexiones. Adicionalmente, se entrenó una red en la que se aplicó abandono al final de cada bloque. Ambos experimentos se realizaron para una UNODE de tolerancia $10^3$. Los resultados obtenidos se pueden consultar en la Tablas 6.3a, 6.3a, 6.3a. Al igual que en los experimentos anteriores, se presenta una visualización de la segmentación realizada por los modelos discutidos en esta sección, en la Figura 6.11.

### 6.4.5. Segmentaciones con mayor y con menor Dice Score

Como parte del análisis, también se muestran las segmentaciones con mayor y con peor Dice Score, para encontrar si la red tiene dificultad para segmentar imágenes con alguna peculiaridad (ver Figuras 6.12, 6.13).

| **Forma 1** | UNODE | ResUNet | U-Net |
|---|---|---|---|
| Accuracy | **0.980** | 0.980 | 0.980 |
| Dice | **0.789** | 0.770 | 0.769 |
| IoU | **0.684** | 0.655 | 0.656 |
| Precision | 0.766 | **0.787** | 0.784 |
| Recall | **0.828** | 0.764 | 0.769 |

(a) Forma 1

| **Forma 2** | UNODE | ResUNet | U-Net |
|---|---|---|---|
| Accuracy | **0.980** | 0.979 | 0.979 |
| Dice | **0.838** | 0.816 | 0.816 |
| IoU | **0.727** | 0.695 | 0.696 |
| Precision | 0.813 | **0.8354** | 0.832 |
| Recall | **0.879** | 0.810 | 0.817 |

(b) Forma 2

| **Forma 3** | UNODE | ResUNet | U-Net |
|---|---|---|---|
| Accuracy | **0.980** | 0.980 | 0.980 |
| Dice | **0.830** | 0.816 | 0.816 |
| IoU | **0.709** | 0.689 | 0.689 |
| Precision | 0.792 | **0.827** | 0.818 |
| Recall | **0.871** | 0.805 | 0.814 |

(c) Forma 3

Tabla 6.2: Comparación entre la UNODE y distintas arquitecturas.

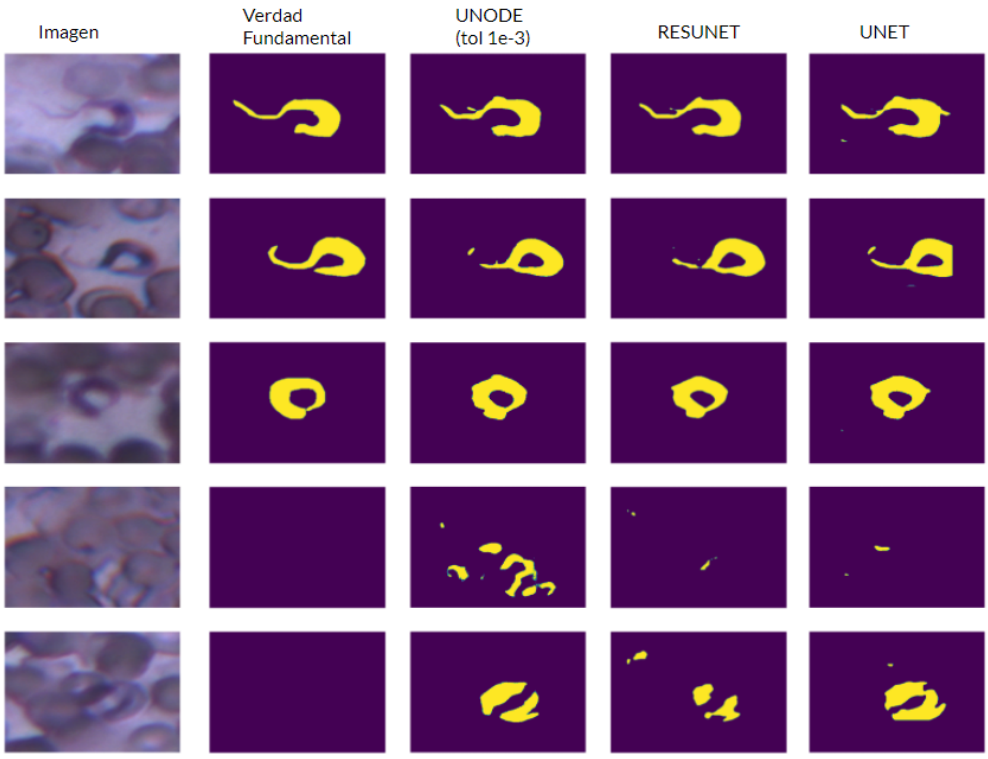



Figura 6.10: Segmentaciones realizadas por diferentes arquitecturas.

| **Forma 1** | UNODE | UNODE Au | UNODE Ab |
|---|---|---|---|
| Accuracy | **0.980** | 0.980 | 0.973 |
| Dice | 0.789 | **0.792** | 0.757 |
| IoU | 0.684 | **0.687** | 0.639 |
| Precision | **0.766** | 0.755 | 0.685 |
| Recall | 0.828 | 0.845 | **0.863** |

(a) Forma 1

| **Forma 2** | UNODE | UNODE Au | UNODE Ab |
|---|---|---|---|
| Accuracy | **0.980** | 0.980 | 0.975 |
| Dice | 0.838 | **0.840** | 0.804 |
| IoU | 0.727 | **0.730** | 0.679 |
| Precision | **0.813** | 0.801 | 0.727 |
| Recall | 0.879 | 0.897 | **0.916** |

(b) Forma 2

| **Forma 3** | UNODE | UNODE Au | UNODE Ab |
|---|---|---|---|
| Accuracy | **0.980** | 0.980 | 0.975 |
| Dice | 0.830 | **0.834** | 0.804 |
| IoU | 0.709 | **0.715** | 0.679 |
| Precision | **0.792** | 0.786 | 0.727 |
| Recall | 0.871 | 0.887 | **0.916** |

(c) Forma 3

Tabla 6.3: Comparación de aumento de datos (Au) y abandono (Ab).

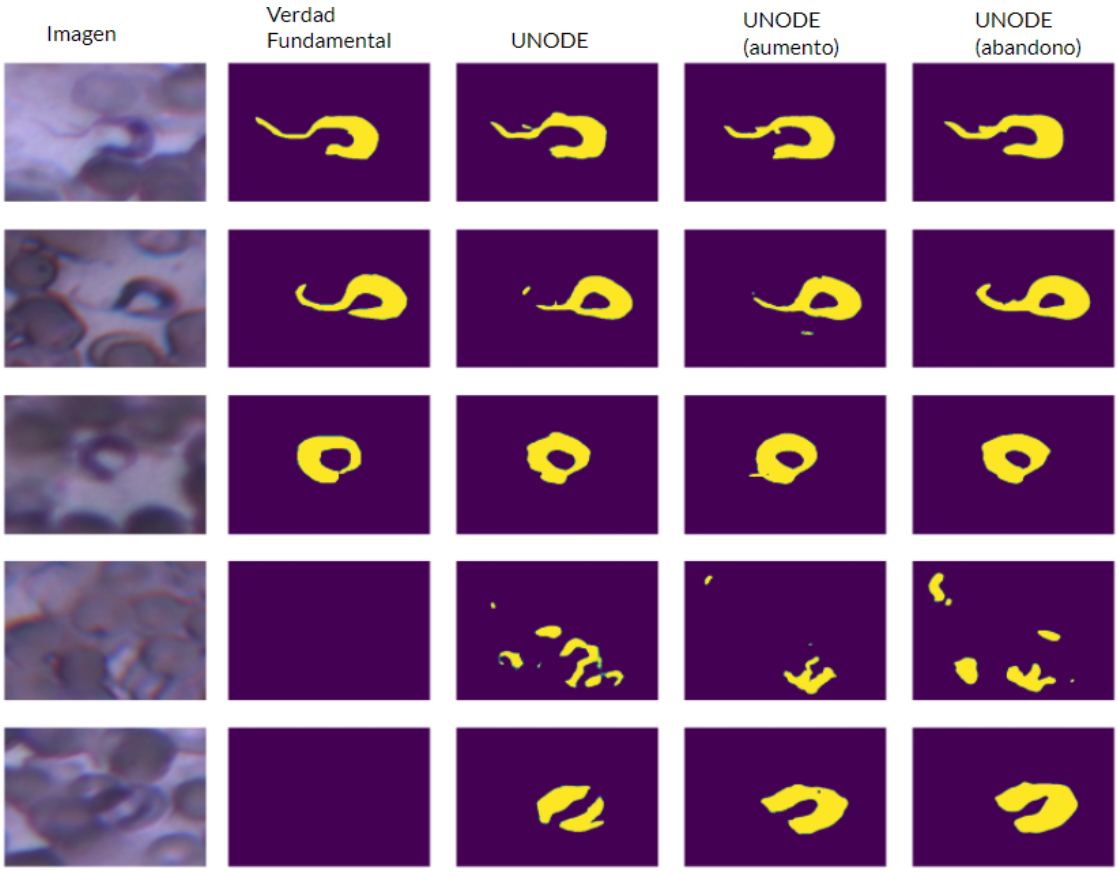



Figura 6.11: Segmentación realizada por versiones de la UNODE (tolerancia $10^{-3}$) sin regularización, con aumento de datos y con abandono.

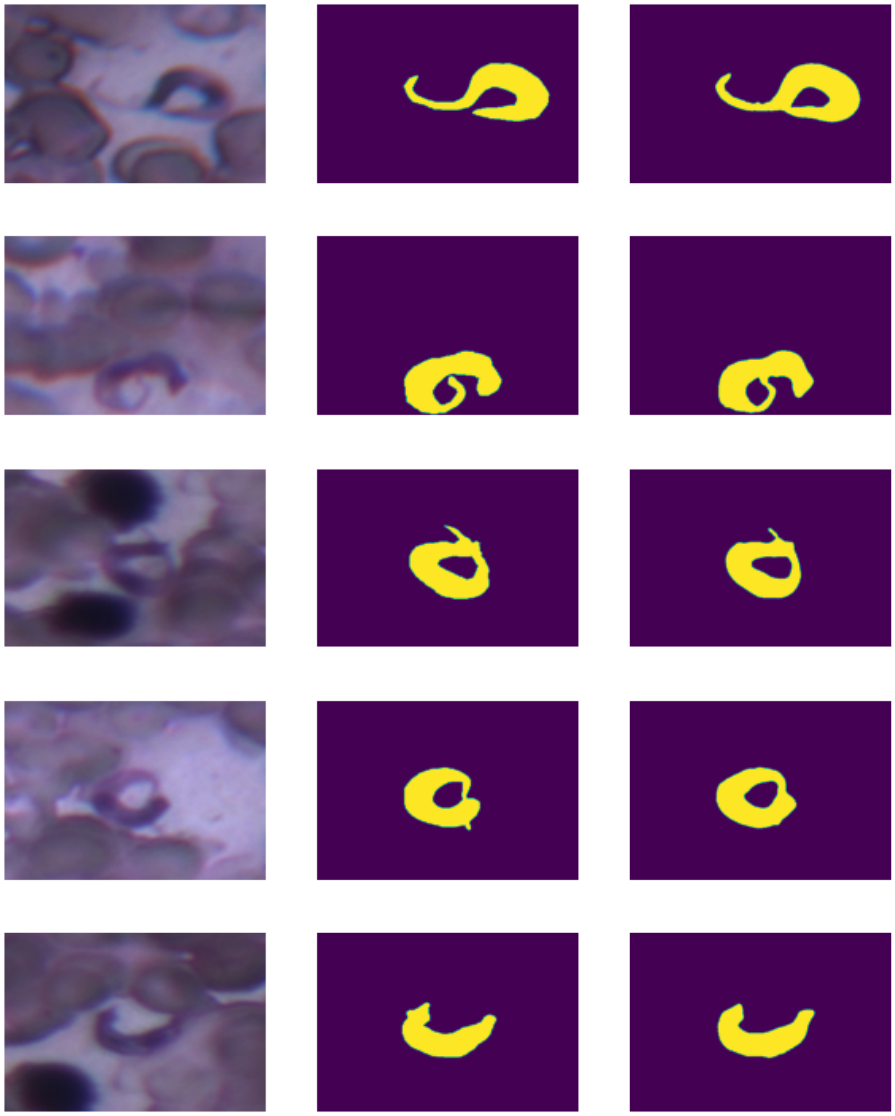


Figura 6.12: Segmentaciones con mayor Dice Score con la UNODE con aumento de datos.

Figura 6.13: Segmentaciones con menor Dice Score con la UNODE con aumento de datos.

| | Accuracy | Dice | IoU | Precision | Recall |
|---|---|---|---|---|---|
| UNODE | 0.981 | 0.790 | 0.685 | 0.766 | 0.829 |
| ResUNet | 0.980 | 0.770 | 0.655 | 0.787 | 0.764 |
| U-Net | 0.980 | 0.769 | 0.656 | 0.784 | 0.769 |
| UNODE Au | **0.981** | 0.792 | 0.688 | 0.755 | 0.846 |
| UNODE Ab | 0.974 | 0.758 | 0.640 | 0.686 | 0.864 |
| Res2UNet | - | **0.840** | **0.730** | **0.850** | 0.820 |
| Gaussiano | - | 0.220 | 0.120 | 0.130 | 0.800 |
| SVM + SP | - | 0.130 | 0.070 | 0.070 | **0.940** |

Tabla 6.4: Comparativa general.

### 6.4.6. Comparativa General

Para finalizar, expondremos una tabla comparativa entre los métodos entrenados aquí, y los que existen en el estado del arte (SVM con súper pixeles y un clasificador gaussiano), esta comparación se puede ver en la Tabla 6.4; en ésta, se toman como referencia las métricas con la forma 1.

## 6.5. Discusión

Se distingue, que con tolerancia $10^{-2}$ se obtuvo un peor desempeño para todas las métricas proporcionadas, lo cuál es algo esperado. Por otro lado, no se consiguieron mejores resultados con una tolerancia de $10^{-4}$ que con $10^{-3}$. Esto sugiere que hay probablemente una tolerancia entre $10^{-3}$ y $10^{-4}$ que es óptima para este problema, y como corolario, que no en todos los casos reducir la tolerancia incrementará el desempeño de la red. En general la UNODE con tolerancia $10^{-3}$ exhibe mejores resultados en todas las métricas (y en todas las formas) propuestas.

En cuanto a la comparativa con diferentes arquitecturas, la UNODE supera en todas las métricas a la ResUNet y a la U-Net, con excepción de la Precision, en la que se ve

superada por ambas redes, siendo la ResUNet la que tuvo mejor desempeño de las dos en esta métrica.

De las imágenes con peor Dice Score, se logra percibir que el modelo presenta dificultades cuando hay más de un parásito en la imagen. También, en algunas imágenes, todos los modelos discutidos confunden en ciertas ocasiones, células muy obscuras con parásitos; esto podría resolverse etiquetando las células en las imágenes y abordar el problema de la segmentación con múltiples clases.

Entre los métodos de regularización, tanto aumento de datos como abandono, consiguieron incrementar el Recall considerablemente, particularmente abandono consiguió 0.916 de Recall. Por otro lado, ambos métodos redujeron la precisión. Finalmente, únicamente el aumento de datos logró incrementar el Dice Score.

## 6.6. Conclusiones

La UNODE y otras arquitecturas basadas en Neural ODEs, son alternativas interesantes a las Neural ODEs por sus principales bondades: entrenamiento con complejidad espacial $O(1)$ en función de la profundidad de la red, y el poder cambiar precisión por tiempo de entrenamiento según se requiera. La investigación de este tipo de redes en aplicaciones reales es muy reducida a la fecha y en esta tesis se aportan evidencias que apuntan a que estos modelos a pesar de que conllevan un tiempo de entrenamiento mayor, bajo ciertas condiciones, no sacrifica eficacia en las métricas evaluadas, y aporta características que podrían ser útiles en ciertas circunstancias.

En este trabajo, se analizó la efectividad de las Ecuaciones Diferenciales Ordinarias Neuronales para la clasificación del parásito *T. Cruzi*. Se analizó el comportamiento de la UNODE con diferentes niveles de tolerancia en la ecuación diferencial que gobierna a esta rede. Se comparó la UNODE con otros modelos, y se alcanzó un Dice Score de 0.79, únicamente superado por la Res2UNet.

Por último, desde el punto de vista práctico, se intentó solucionar un posible sobreajuste de los datos observado en la UNODE, mediante aumento de datos y abandono, hallando

mejoras únicamente con el aumento de datos.

### 6.6.1. Futuros Trabajos

En los últimos años se ha investigado extensamente en la emergente área del "**Aprendizaje profundo continuo**". Se han implementado ideas conocidas en el contexto de sistemas dinámicos a las Neural ODEs, que pueden mejorar la estabilidad, velocidad, entre otras cosas. Estos trabajos, pueden ayudar a construir mejores Neural ODEs, y que a su vez podrían ayudar en sus aplicaciones, como es el caso de la UNODE. Aunque la diversidad de artículos al respecto es muy amplia, listaré a continuación, los que considero que podrían tener una repercusión inmediata.

Por ejemplo, en [45–47] involucran ecuaciones diferenciales estocásticas (SGDs) en lugar de ODEs. En particular [45], utiliza este paradigma para con el fin de estabilizar las ODEs con ruido estocástico, lo cual podría implementarse directamente con la UNODE para conseguir mejores resultados.

En [42], presentan generalizaciones de las Neural ODEs, de las cuales, algunas consiguieron mejorar la Accuracy en MNIST y CIFAR, que son dos dataset muy conocidos, y utilizados ampliamente para estudiar resultados generales.

Uno de los restos a la hora de trabajar con las Neural ODEs, es el tiempo que tarda en entrenarse, lo cual a su vez, introduce dificultades para probar diferentes configuraciones e hiper-parámetros. Los artículos [48–50], se enfocan en diferentes métodos para aumentar la velocidad del entrenamiento.

# Bibliografía


[1] Julie Clayton. Chagas disease 101. *Nature*, 465(7301):S4–S5, Jun 2010.

[2] Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural ordinary differential equations, 2018.

[3] Lev Semenovich Pontryagin, V G Boltyanskii, R V Gamkrelidze, and E F Mishchenko. *The mathematical theory of optimal processes*. Wiley, New York, NY, 1962.

[4] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation, 2015.

[5] Victor Uc-Cetina, Carlos Brito-Loeza, and Hugo Ruiz-Piña. Chagas parasites detection through gaussian discriminant analysis. *Abstraction & Application*, 8:6–17, 08 2013.

[6] Roger Soberanis-Mukul, Víctor Uc-Cetina, Carlos Brito-Loeza, and Hugo Ruiz-Piña. An automatic algorithm for the detection of trypanosoma cruzi parasites in blood sample images. *Computer methods and programs in biomedicine*, 112, 09 2013.

[7] Víctor Uc-Cetina, Carlos Brito-Loeza, and Hugo Ruiz-Piña. Chagas parasite detection in blood images using adaboost. *Computational and Mathematical Methods in Medicine*, 2015:1–13, 04 2015.

[8] Roger David Soberanis Mukul. Algoritmos de segmentación de trypanosoma cruzi en imagenes de muestras sanguineas. Master's thesis, Universidad Autónoma de Yucatán, 2014.

[9] Allan Ojeda-Pat, Anabel Martin-Gonzalez, and Roger Soberanis-Mukul. *Convolutional Neural Network U-Net for Trypanosoma cruzi Segmentation*, pages 118–131. Springer International Publishing, 03 2020. Third International Symposium, ISICS 2020.

[10] Aimon Rahman, Hasib Zunair, M Sohel Rahman, Jesia Quader Yuki, Sabyasachi Biswas, Md Ashraful Alam, Nabila Binte Alam, and M. R. C. Mahdy. Improving malaria parasite detection from red blood cell using deep convolutional neural networks, 2019.

[11] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[12] Julisa Abraham. Malaria parasite segmentation using u-net: Comparative study of loss functions. *Communications in Science and Technology*, 4:57–62, 12 2019.

[13] Ava P. Soleimany, Harini Suresh, Jose Javier Gonzalez Ortiz, Divya Shanmugam, Nil Gural, John Guttag, and Sangeeta N. Bhatia. Image segmentation of liver stage malaria infection with spatial uncertainty sampling, 2019.

[14] Marc Górriz, Albert Aparicio, Berta Raventós, Verónica Vilaplana, Elisa Sayrol, and Daniel López-Codina. Articulated motion and deformable objects. *Lecture Notes in Computer Science*, 2018.

[15] Hans Pinckaers and Geert Litjens. Neural ordinary differential equations for semantic segmentation of individual colon glands, 2019.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[17] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, New York, NY, USA, second edition, 2006.

[18] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[19] A 2021 guide to semantic segmentation. https://nanonets.com/blog/semantic-image-segmentation-2020/. Accessed: 2021-05-25.

[20] A 2021 guide to semantic segmentation. https://heartbeat.fritz.ai/a-2019-guide-to-semantic-segmentation-ca8242f5a7fc. Accessed: 2021-05-25.

[21] Semantic segmentation algorithm is now available in amazon sagemaker. https://aws.amazon.com/es/blogs/machine-learning/semantic-segmentation-algorithm-is-now-available-in-amazon-sagemaker/. Accessed: 2021-10-20.

[22] An overview of semantic image segmentation. https://www.jeremyjordan.me/semantic-segmentation/. Accessed: 2021-10-08.

[23] Allan Eduardo Ojeda Pat. Res2unet: Una red completamente convolucional para la segmentación del parásito trypanosoma cruzi. Master's thesis, Universidad Autónoma de Yucatán, 2020.

[24] Shruti Jadon. A survey of loss functions for semantic segmentation. *2020 IEEE Conference on Computational Intelligence in Bioinformatics and Computational Biology (CIBCB)*, Oct 2020.

[25] Carole H. Sudre, Wenqi Li, Tom Vercauteren, Sebastien Ourselin, and M. Jorge Cardoso. Generalised dice overlap as a deep learning loss function for highly unbalanced segmentations. *Lecture Notes in Computer Science*, page 240–248, 2017.

[26] Lars Nieradzik. Loss functions for segmentation. https://lars76.github.io/2018/09/27/loss-functions-for-segmentation.html. Accessed: 2021-05-25.

[27] Loss function reference for keras & pytorch. https://www.kaggle.com/bigironsphere/loss-function-library-keras-pytorch. Accessed: 2021-05-25.

[28] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016.

[29] Ehsan Fathi and Babak Maleki Shoja. Chapter 9 - deep neural networks for natural language processing. In Venkat N. Gudivada and C.R. Rao, editors, *Computational Analysis and Understanding of Natural Languages: Principles, Methods and Applications*, volume 38 of *Handbook of Statistics*, pages 229–316. Elsevier, 2018.

[30] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016.

[31] Convolutional neural networks (cnns): An illustrated explanation. https://blog.xrds.acm.org/2016/06/convolutional-neural-networks-cnns-illustrated-explanation/. Accessed: 2021-10-09.

[32] ¿qué es max pooling en cnn? https://es.quora.com/Qu%C3%A9-es-Max-Pooling-en-CNN. Accessed: 2021-10-09.

[33] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[34] How data augmentation impacts performance of image classification, with codes. https://analyticsindiamag.com/image-data-augmentation-impacts-performance-of-image-classification-with-codes/. Accessed: 2021-10-08.

[35] Data augmentation for semantic segmentation – deep learning. https://idiotdeveloper.com/data-augmentation-for-semantic-segmentation-deep-learning/. Accessed: 2021-10-08.

[36] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6:107–116, 04 1998.

[37] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR.

[38] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification, 2015.

[39] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[40] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation, 2015.

[41] Zhengxin Zhang, Qingjie Liu, and Yunhong Wang. Road extraction by deep residual u-net. *IEEE Geoscience and Remote Sensing Letters*, 15(5):749–753, May 2018.

[42] Stefano Massaroli, Michael Poli, Jinkyoo Park, Atsushi Yamashita, and Hajime Asama. Dissecting neural odes, 2021.

[43] Deriving the adjoint equation for neural odes using lagrange multipliers. https://math.stackexchange.com/questions/3608614/method-of-adjoints-neural-odes. Accessed: 2021-08-316.

[44] Method of adjoints, neural odes. https://vaipatel.com/deriving-the-adjoint-equation-for-neural-odes-using-lagrange-multipliers/. Accessed: 2021-08-16.

[45] Xuanqing Liu, Tesi Xiao, Si Si, Qin Cao, Sanjiv Kumar, and Cho-Jui Hsieh. Neural SDE: stabilizing neural ODE networks with stochastic noise. *CoRR*, abs/1906.02355, 2019.

[46] Junteng Jia and Austin R. Benson. Neural jump stochastic differential equations. *CoRR*, abs/1905.10403, 2019.

[47] Xuechen Li, Ting-Kam Leonard Wong, Ricky T. Q. Chen, and David Duvenaud. Scalable gradients for stochastic differential equations. *CoRR*, abs/2001.01328, 2020.

[48] Chris Finlay, Jörn-Henrik Jacobsen, Levon Nurbekyan, and Adam M Oberman. How to train your neural ode: the world of jacobian and kinetic regularization, 2020.

[49] Jacob Kelly, Jesse Bettencourt, Matthew James Johnson, and David Duvenaud. Learning differential equations that are easy to solve, 2020.

[50] Patrick Kidger, Ricky T. Q. Chen, and Terry Lyons. "hey, that's not an ode": Faster ode adjoints via seminorms, 2021.

[51] Enfermedad de chagas. https://www.cdc.gov/parasites/chagas/es/. Accessed: 2021-10-28.

[52] Enfermedad de chagas. Our U-net wins two Challenges at ISBI 2015. Accessed: 2021-11-05.