UNIVERSIDAD DE BUENOS AIRES
Facultad de Ciencias Exactas y Naturales
Departamento de Matemática

Tesis de Licenciatura

Un enfoque probabilístico de las redes neuronales aplicadas a la clasificación multinomial de imágenes.

Maite Lucía Angel

Directores: Dra. Mariela Sued y Dr. Andrés Farall.

2021

Esta tesis está dedicada a mi abuelo Omar, mi abuela Cusa,
mi tío Héctor y mi madrina Alicia.

.

Índice general

1. Introducción **5**

2. Problemas de clasificación multinomial **7**
2.1. Fundamentos de la clasificación multinomial 7
2.1.1. Predicciones. 9
2.2. Tipos de clasificadores 9

3. Reconocimiento de expresiones faciales **11**

4. Regresión Logística **14**
4.1. Clasificación binaria 14
4.2. Clasificación multiclase 16

5. Redes Neuronales Fully Connected **19**
5.1. Fundamentos de las redes neuronales 19
5.1.1. Perceptrón y el modelo logístico 20
5.1.2. Projection Pursuit Regression 21
5.1.3. Estructura general 23
5.1.4. Adaptación para resultados multinomiales 25
5.2. Entrenamiento de la red. 26
5.2.1. Descenso por el gradiente (GD) 28
5.2.2. Descenso por el gradiente estocástico 29
5.2.3. Back propagation 30
5.3. Predicciones 30
5.3.1. Elección función de activación 30
5.4. Laboratorio Redes Neuronales 32

6. Redes Neuronales Convolucionales **37**
6.1. Fundamentos de las redes neuronales convolucionales 37
6.1.1. Filtros y feature maps 37
6.1.2. Pooling 40

6.1.3. Estructura red convolucional 42
6.1.4. Nociones de entrenamiento 43
6.1.5. Predicciones 43
6.2. Ventajas y propiedades de las arquitecturas convolucionales . . 44
6.3. Laboratorio CNN 45

7. Discusión: Calibración de los modelos 55

8. Resultados Reconocimiento de Expresiones 58
8.1. Procesamiento de materiales 58
8.2. Métodos y métricas 59
8.3. Exploración de resultados 63

9. Conclusiones 69

Bibliografía 71

Apéndice 73

Capítulo 1

Introducción

El *aprendizaje estadístico*, como lo llaman los autores Hastie y Tibshirani en el libro "An introduction to Statistical Learning with Applications in R", es una área de considerable reciente desarrollo de la estadística que tiene una relación intrínseca con descubrimientos de la ciencia de la computación, en particular con el *aprendizaje automático*. Con la explosión del fenómeno de grandes volúmenes de datos, este campo comenzó a ser aplicado en contextos muy diversos tales como las finanzas, la biología, la meteorología y la detección de enfermedades.

En particular, en este estudio, nos concentramos en un *problema supervisado*, más específicamente, en la *clasificación multinomial*. y nos interesará hacer una revisión con cierto enfoque probabilístico de una técnica específica de este área: *las redes neuronales clásicas* y *convolucionales*.

Por su parte el análisis supervisado puede definirse como la construcción de reglas con el objetivo de predecir el valor de una respuesta correspondiente a ciertas variables predictoras, a partir de datos. Se lo denomina de esta forma ya que se conoce la respuesta para cada una de las observaciones. El proceso que un algoritmo aprenda de los datos puede pensarse como un profesor *supervisando* el proceso de aprendizaje.

El caso particular de clasificación se da cuando la respuesta es una variable categórica, por ejemplo "enfermo"/"no enfermo" ó "rojo"/"verde"/"azul". Será *binaria* cuando la cantidad de categorías sea exactamente 2 (codificadas como 0 y 1) y será *multinomial* cuando sea al menos 3 (codificadas nuevamente como 0,1,2,...).

La tarea puntual a resolver será asignar a cada realización de las variables explicativas un vector en el $[0,1]^J$, donde J será el número de posibles etiquetas, tal que cada coordenada representa la probabilidad estimada de pertenecer a la correspondiente clase. Pediremos entonces que la relación entre ellas sea que sumen 1. Enunciaremos y seguiremos el criterio de Bayes

de forma que se asignará cada nueva observación a la categoría con mayor probabilidad estimada.

En este contexto, las redes neuronales surgen como un estimador de dicha probabilidad. Se les ha atribuido, en muchas ocasiones, que su mayor impacto viene de haber revitalizado el campo del *reconocimiento de patrones*. Este último es la ciencia que se ocupa de procesos de clasificación de objetos con el propósito de extraer información que permita establecer distintas propiedades y características de ellos. Es usualmente aplicado a la *minería de datos*, *reconocimiento de escritura a mano*, *recuperación de datos multimedia* y *reconocimiento de rostros* entre otros.

Usaremos como ejemplo de aplicación el problema de *reconocimiento de expresiones faciales* que implica el análisis de imágenes como input para reconocer la presencia de un sentimiento.

El presente trabajo de tesis se organiza como sigue: en el *Capítulo 2* presentaremos una introducción teórica al problema de clasificación. En el *Capítulo 3* daremos una breve descripción del problema de reconocimiento de expresiones faciales que servirá de guía a lo largo de todo el documento. En el *Capítulo 4* estudiaremos la discriminación logística en su versión binaria y multinomial. En el *Capítulo 5* estudiaremos las redes neuronales Fully Connected, sus fundamentos y entrenamiento. En el *Capítulo 6* introduciremos las redes neuronales convolucionales, sus elementos claves y metodología. En el *Capítulo 7* discutiremos brevemente la calibración de los modelos tratados en el estudio.En el *Capítulo 8* analizaremos los resultados de las técnicas de los métodos descritos anteriormente en el problema de clasificación de imágenes. Por último en el *Capítulo 9* daremos las conclusiones y los posibles pasos a seguir posteriores a este trabajo.

Luego de cada capítulo se detallarán los códigos en Python que implementen las técnicas mencionadas en un ambiente controlado.

Capítulo 2

Problemas de clasificación multinomial

En los problemas de clasificación se cuenta con un vector aleatorio (X, Y) con $X \in \mathbb{R}^d$ e $Y \in E$ donde $E = \{E_0, ..., E_{J-1}\}$ será el conjunto de las J etiquetas posibles.

Un *clasificador* será una regla de decisión que dado una realización de X le asigna una etiqueta en E. Lo notamos como la función $C : \mathbb{R}^d \to E$.

Por ejemplo *el reconocimiento de expresiones faciales*, que trataremos en este estudio, es la tarea que implica clasificar imágenes de rostros en distintas categorías como "enojo" o "felicidad".

Este es un clásico problema de clasificación donde en el rol de las observaciones (X) tendremos imágenes y las posibles respuestas (Y) serán las distintas expresiones.

2.1. Fundamentos de la clasificación multinomial

A la hora de pensar en cómo armar un buen clasificador se suele pensar en cómo obtener la menor cantidad de clasificaciones erróneas posibles. Cabe aclarar que la posibilidad de que una observación no pertenezca a ninguna de las clases no estará considerada en este estudio.

Definimos entonces como representante del **error/pérdida** del clasificador C a $L(C) = P(C(X) \neq Y)$.

Luego un clasificador óptimo quedará definido como:

$$C_{opt} = \underset{C}{argmin}\ \ L(C) = \underset{C}{argmin}\ \ P(C(X) \neq Y)$$

La solución a este problema de optimización vendrá dada por un clasificador muy simple que toma la idea de asignar cada observación a la *clase más probable dado su valor*, en otras palabras clasificar una observación x a la clase E_j de modo que la probabilidad que $P(Y = E_j|X = x)$(probabilidad a posteriori) sea máxima.

A esta herramienta se la conoce como *clasificador de Bayes* y se define de la siguiente forma:

$$C_{Bayes}(x) = \underset{E_j \in E}{argmax}\ \ P(Y = E_j|X = x)$$

El siguiente resultado muestra que C_{Bayes} es óptimo, en el sentido de minimizar el error de clasificación.

Teorema:
Dado (X, Y) vector aleatorio, luego $C_{Bayes} = \underset{C}{argmin}\ \ P(C(X) \neq Y)$.
Demostración:
Para fijar ideas, haremos la demostración asumiendo que (X, Y) es un vector discreto.

Primero notemos que utilizando el **Teorema de Bayes**:

$$P(Y = E_j|X = x) = \frac{P(X = x|Y = E_j)P(Y = E_j)}{P(X = x)}$$

Luego C_{Bayes} cumple que $C_{Bayes} = \underset{E_j \in E}{argmax} P(X = x|Y = E_j)P(Y = E_j)$.

Sea C un clasificador cualquiera:

$$L(C) = P(C(X) \neq Y) = 1 - P(C(X) = Y)$$

$$= 1 - \sum_{E_j \in E} P(C(X) = Y|Y = E_j)P(Y = E_j)$$

$$= 1 - \sum_{E_j \in E} P(C(X) = E_j|Y = E_j)P(Y = E_j)$$

Si llamamos q_j a la función de probabilidad puntual $X|Y = E_j$ tenemos que:

$$= 1 - \sum_{j=0}^{J-1} \left\{ \sum_{\{x \in \mathbb{R}^d : C(x) = E_j\}} q_j(x) \right\} P(Y = E_j)$$

$$= 1 - \sum_{x \in \mathbb{R}^d} \sum_{j=0}^{J-1} q_j(x)\, P(Y = E_j) I_{(C(x)=E_j)}$$

Luego para cada x tenemos que:

$$\sum_{j=0}^{J-1} q_j(x) P(Y = E_j) I_{(C(x)=E_j)} \leq \max_{E_j \in E} \; q_j(x) P(Y = E_j) = \sum_{j=0}^{J-1} q_j(x) P(Y = E_j) I_{(C_{Bayes}(x)=E_j)}$$

de donde concluimos que $L(C) \geq L(C_{Bayes})$.

□

2.1.1. Predicciones.

Hemos encontrado una caracterización de C_{Bayes}, el clasificador que minimiza el error de clasificación. Recordemos que admite las siguientes dos representaciones:

$$C_{Bayes}(x) = \underset{E_j \in E}{argmax} \;\; P(Y = E_j | X = x)$$

y

$$C_{Bayes}(x) = \underset{E_j \in E}{argmax} P(X = x | Y = E_j) P(Y = E_j)$$

Estas resultan inspiradoras de muchos de los métodos de clasificación que se construyen utilizando datos. Es decir, teniendo dichos resultados teóricos podremos hacer uso de un criterio de plug-in de forma tal que, basados en observaciones, estimaremos las probabilidades de la escritura de Bayes elegida.

Por ejemplo para la primera escritura de Bayes, donde debemos estimar la $P(Y = E_j | X = x)$, que llamaremos $p_j(x)$, nos armaremos $\hat{p_j}(x)$ a partir de datos para estimar dicha probabilidad.

Luego tomaremos la decisión de predecir la etiqueta según el máximo $\hat{p_j}(x)$. En particular en el problema binario, $j \in \{0, 1\}$, seleccionar la clase de mayor valor equivale a decidirse por la etiqueta E_j tal que $\hat{p_j}(x) > 0{,}5$.

2.2. Tipos de clasificadores

De acuerdo a las dos representaciones de Bayes expuestas, existen dos grandes grupos que suelen presentarse en la literatura: los discriminativos y los generativos. Por otra parte, cada una de estas categorías admite métodos paramétricos y no paramétricos.

Cuando trabajamos con métodos paramétricos, modelamos utilizando una expresión conformada por un conjunto de coeficientes que se aprenderán de los datos, en lo que se conoce como entrenamiento del modelo. Esta estructura tiende a propiciar su fácil interpretación y suelen necesitar pocos datos por lo que tienden a ser muy rápidos. Por otro lado son rígidos, es decir, funcionan muy bien cuando el modelo de probabilidad supuesto es el correcto, de lo contrario puede dificultarse la tarea de aproximar la función subyacente del mismo.

En contraposición, los métodos no paramétricos típicamente hacen apenas suposiciones de regularidad sobre la función que procuran estimar y usan patrones presentes en los puntos cercanos. Esto hace que sean muy flexibles pero requieren de una gran cantidad de datos para un buen funcionamiento (suelen ser un poco lentos) y son mucho más propensos a parecerse demasiado a los datos (overfitting). A modo de ejemplo, presentamos algunos métodos en cada una de las clases mencionadas:

- ***Modelos Discriminativos***: Estos son los que efectivamente modelan $P(Y = E_j|X = x)$.Algunos ejemplos clásicos son la regresión logística como modelo paramétrico mientras que KNN (K vecinos más cercanos) es una propuesta NO paramétrica.

- ***Modelos Generativos***: Esta clase de modelos se encargan de modelar la distribución de $X|Y = E_j$.Algunos ejemplos de esta categoría son LDA (linear discriminant analysis) como modelo paramétrico y Naive Bayes como NO paramétrico.

En los siguientes capítulos estudiaremos los modelos basados en **redes neuronales clásicas** y **convolucionales**. Fijada la arquitectura de la red, veremos que cada uno de ellos puede ser considerado como un clasificador discriminativo paramétrico. Sin embargo, la riqueza en la elección de la arquitectura hace que el mundo de las redes, en su conjunto, se lo considere dentro de los métodos NO paramétricos. Resultan modelos sumamente potentes con una numerosa cantidad de parámetros que en la práctica han demostrado un excelente desempeño.

Capítulo 3

Reconocimiento de expresiones faciales

El reconocimiento de expresiones faciales es la tarea que implica clasificar imágenes de rostros en distintas expresiones como "enojo" o "felicidad".

Desde el punto de vista computacional, el problema reside en crear un algoritmo que comprenda los patrones característicos de cada sentimiento en el rostro humano y lo interprete como tal.

Dataset FER2013

Para introducir distintos conceptos y formalizar la idea de construir una solución al problema en cuestión utilizaremos el dataset FER-2013 (Facial Expression Recognition 2013) creado por Pierre-Luc Carrier y Aaron Courville. Usaremos una versión preliminar que hicieron pública en la competencia de Kaggle *Challenges in Representation Learning: Facial Expression Recognition Challenge*.

Origen y procesamiento

Describiremos en síntesis los pasos que se utilizaron para conseguir el dataset:

- Usando la aplicación de imágenes de Google se buscaron 184 palabras claves en relación a distintas emociones. Se las combinaron con otras palabras relacionadas a género, edad y etnia, de modo que se obtuvieron 600 sentencias usadas como queries (consultas de búsqueda). Se seleccionaron las primeras 1000 imágenes de cada una.

- Utilizando OpenCV (biblioteca libre de visión artificial) se obtuvieron

las **bounding boxes** (delimitadores) alrededor de cada una de las caras presentes en las fotos recolectadas y una clasificación provisoria de las imágenes.

- Se realizó un trabajo manual de re-etiquetado y re-encuadramiento de las imágenes, cuando fue necesario. Las imágenes fueron procesadas de modo que la cara esté centrada y ocupen todas aproximadamente la misma cantidad de espacio.
- Se removieron imágenes duplicadas
- Las imágenes fueron reformadas a 48x48 píxeles y pasadas a escala de grises.

Mehdi Mirza y Ian Goodfellow prepararon la versión preliminar de este dataset repasando todas las imágenes y mapeandolas en las mismas 7 categorías de expresiones faciales que presentamos a continuación.

Cantidad de imágenes

El dataset consiste en:

- Un conjunto de datos de entrenamiento (notado como *Training*) que contiene 28.709 ejemplos de imágenes etiquetadas.
- Un conjunto de datos de testing público (notado como *PublicTest*) que contiene 3.589 ejemplos de imágenes etiquetadas.
- Un conjunto de datos de testing privado (notado como *PrivateTest*) que contiene 3.589 ejemplos de imágenes etiquetadas.

Dada la naturaleza de estos datos (pensado para competencias) encontramos sets de testing públicos y privados que actúan como conjuntos de validación y testeo final para comparar los distintos modelos que apliquen.

Emociones

Las etiquetas alusivas a la expresión facial de la imagen que la clasifican son:

- 0-Enojo: *4953 imágenes*
- 1-Disgusto: *547 imágenes*
- 2-Miedo: *5121 imágenes*

- 3-Felicidad: *8989 imágenes*
- 4-Tristeza: *6077 imágenes*
- 5-Sorpresa: *4002 imágenes*
- 6-Neutral: *6198 imágenes*

Veamos la siguiente imagen:

Figura 3.1: "Samuel Jackson enojado"

El ojo humano puede interpretarla sin mayor dificultad: "esa es una expresión de enojo", expresamos sin detenernos a pensar en lo complicada de esta tarea.

Representación de los datos

- Cada imagen será representada como una matriz, donde cada celda será un número real, en este caso por números naturales entre el 0 (píxel negro) y el 255 (píxel blanco) que representará "qué tan gris" es ese píxel.
- Cada una de ellas tendrá asociada una *única* etiqueta numerada entre 0-6 según la emoción que corresponda.

Formalmente, cada dato es un par (x, y) donde $x \in \mathbb{R}^{48\times48}$, $y \in E = \{0, 1, 2, 3, 4, 5, 6\}$.

Capítulo 4

Regresión Logística

Antes de pasar al siguiente capítulo y describir nuestra técnica de interés nos tomaremos un momento para estudiar la *discriminación logística*. Este es un método discriminativo paramétrico. La principal atracción de esta forma de clasificar es que el modelo resulta ser muy interpretable: su capacidad explicativa permite conocer la influencia de las variables predictoras en la predicción. Daremos una introducción teórica del método para la clasificación binaria y para la multiclase, siguiendo el trabajo de Agresti, donde se puede encontrar un abordaje exhaustivo de estos modelos.

4.1. Clasificación binaria

Por simplicidad, comencemos pensando en un problema de clasificación binaria ($Y \in \{E_0 = 0, E_1 = 1\}$).

La principal hipótesis que impone el enfoque logístico es asumir que para ciertos $\alpha \in \mathbb{R}^d, \beta \in \mathbb{R}$:

$$P(Y = E_1|X) = p_1(X; (\alpha, \beta)) \text{ , con } \quad p_1(x; (\alpha, \beta)) = \frac{e^{\alpha^T x+\beta}}{1 + e^{\alpha^T x+\beta}} \tag{4.1}$$

Es decir proponemos un predictor lineal $\alpha^T X + \beta$ y con el uso de la función sigmoidea $S(x) = \frac{e^x}{1+e^x}$ obtenemos un valor perteneciente al $[0, 1]$ que postulamos como probabilidad.

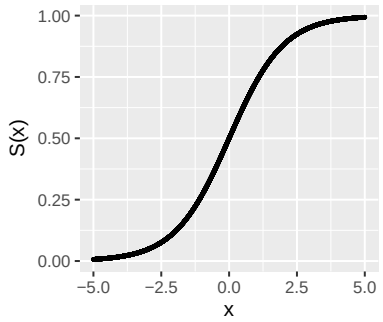


Figura 4.1: "Gráfico Sigmoidea"

Despejando $\alpha^T X + \beta$ de (4.1), obtenemos que:

$$\log\left(\frac{P(Y = E_1|X)}{1 - P(Y = E_1|X)}\right) = \log\left(\frac{P(Y = E_1|X)}{P(Y = E_0|X)}\right) = \alpha^T X + \beta \qquad (4.2)$$

Luego el predictor lineal representa, en escala logarítmica, la diferencia de la probabilidad de pertenecer a cada clase.

Dada una realización $(x^i, E^i), i \in \{1, ..., N\}$ de observaciones $x^i \in \mathbb{R}^d$ con etiquetas $E^i \in E$, podremos estimar $\alpha \in \mathbb{R}^d$ y $\beta \in \mathbb{R}$ con $\hat{\alpha} \in \mathbb{R}^d$ y $\hat{\beta} \in \mathbb{R}$ a través del método de máxima verosimilitud. Del modelo propuesto en (4.1), obtenemos:

$$\log\left(P(Y^1 = E^1, ..., Y^N = E^N \mid X^1 = x^1, ..., X^N = x^N)\right) =$$
$$\log\left(\prod_i P(Y = E^i \mid X = x^i)\right) = \log\left(\prod_{i=1}^{N} p_1(x^i; (\alpha, \beta)^{E^i} (1 - p_1(x^i; (\alpha, \beta)))^{1-E^i}\right) =$$
$$\sum_{i=1}^{N} E^i \log(p_1(x^i; (\alpha, \beta))) + (1 - E^i) \log(1 - p_1(x^i; (\alpha, \beta)))$$

Tenemos entonces que la log verosimilitud ($LV : \mathbb{R}^d \times \mathbb{R} \to \mathbb{R}$) queda definida como:

$$LV(\alpha, \beta) = \sum_i E^i \log\left(\frac{e^{\alpha^T x^i + \beta}}{1 + e^{\alpha^T x^i + \beta}}\right) + (1 - E^i) \log\left(1 - \frac{e^{\alpha^T x^i + \beta}}{1 + e^{\alpha^T x^i + \beta}}\right)$$

Luego buscaremos maximizar esta última expresión obteniendo los estimadores $\hat{\beta}$ y $\hat{\alpha}$ en función de la muestra, sin embargo hacerlo no es una tarea

trivial. Se han implementado varios métodos iterativos para obtener soluciones aproximadas, entre ellos los métodos de Newton-Raphson y descenso por el gradiente. Profundizaremos en este último en las próximas secciones.

Finalmente con los parámetros estimados obtenemos estimadores de las probabilidades que resulten:

$$\hat{p_1}(x) = \frac{e^{\hat{\beta}+\hat{\alpha}^T x}}{1+e^{\hat{\beta}+\hat{\alpha}^T x}}$$

$$\hat{p_0}(x) = 1 - \frac{e^{\hat{\beta}+\hat{\alpha}^T x}}{1+e^{\hat{\beta}+\hat{\alpha}^T x}} = \frac{1}{1+e^{\hat{\beta}+\hat{\alpha}^T x}}$$

Como dijimos antes siguiendo el criterio de plug-in de Bayes, la regla de decisión quedará definida como $C : \mathbb{R}^d \to E$ tal que:

$$C(x) = \begin{cases} 0 & \hat{p_1}(x) \leq 0{,}5 \\ 1 & \text{en caso contrario} \end{cases}$$

4.2. Clasificación multiclase

Veremos la extensión de este último modelo al mundo multi-clase, donde ahora $Y \in E = \{E_0 = 0, .., E_{J-1} = J-1\}$. Siguiendo la propuesta presentada en (4.2), la principal hipótesis ahora será asumir que existen $\alpha_j \in \mathbb{R}^d, \beta_j \in \mathbb{R}$ tales que, para $j \in \{1, ..., J-1\}$, tenemos:

$$\log\left(\frac{P(Y = E_j|X)}{P(Y = E_0|X)}\right) = \alpha_j^T X + \beta_j \ . \tag{4.3}$$

En tal caso, denotando θ el vector con todos los parámetros (α_j, β_j), $j = 1, \ldots, J-1$, deducimos que

$$P(Y = E_j \mid X) = p_j(X;\theta)\,, \quad p_j(x;\theta) = \frac{\exp(\alpha_j^T x + \beta_j)}{1 + \sum_{k=1}^{J-1} \exp(\alpha_k^T x + \beta_k)}\ , \tag{4.4}$$

$$P(Y = E_0 \mid X) = p_0(X;\theta)\,, \quad p_0(x;\theta) = \frac{1}{1 + \sum_{k=1}^{J-1} \exp(\alpha_k^T x + \beta_k)}\ . \tag{4.5}$$

En cuanto a la estimación de los coeficientes se sigue análogamente al caso anterior por el método de máxima verosimilitud. Para definir convenientemente la log-verosimilitud en este caso, reformularemos un poco como expresamos la etiqueta; usaremos la codificación conocida como *one hot*.

En este estudio indexaremos a los vectores de longitud J de forma que sus posiciones resulten ser $0, ..., J-1$ para ser consistentes con la notación de números de etiquetas (inclinándonos hacia el comportamiento del lenguaje computacional).

En esencia, para definir la nueva codificación de las etiquetas, convertiremos cada $E_j \in E$ en un vector $y_j \in \{0,1\}^J$ de forma que en la posición j tenga 1 y el resto sea 0, para $j = 0, \ldots, J-1$. y_j será el j-ésimo vector canónico.

Además, llamamos $y_{j,k}$ a la k-ésima posición del vector y_j. Con esto, si $(p_0, \ldots, p_{J-1})$ denota el vector de probabilidades puntuales de una variable Y tomando valores en $\{E_0, \ldots, E_{J-1}\}$, podemos escribir

$$P(Y = E_j) = P(Y = y_j) = \prod_{k=0}^{J-1} p_k^{y_{j,k}}.$$

Notemos que solo sobrevive un p_k, que es p_j, porque solo la coordenada j del vector y_j es 1.

En las siguientes secciones haremos un abuso de notación y nos referiremos a la etiqueta indistintamente como E y como y, siendo y_j la j-ésima posición del vector y.

Con esta nueva notación, bajo el modelo asumido, tenemos que

$$P(Y = E \mid X = x) = P(Y = y \mid X = x) = \prod_{j=0}^{J-1} p_j(x;\theta)^{y_j} \ ,$$

con $p_j(x;\theta)$ definidos en (4.4) y (4.5), para $j \in \{1, \ldots, J-1\}$ y $j = 0$, respectivamente. Reproduciendo entonces el análisis presentado para el caso de respuesta binaria, podemos expresar la log- verosimilitud LV utilizando el modelo propuesto como

$$LV(\theta) = \sum_{i=1}^{N} \sum_{j=0}^{J-1} y_j^i \log\left(p_j(x^i;\theta)\right) \ .$$

Usando las propiedades de la función logaritmo y recordando que la suma de las coordenadas de y^i vale uno, llegamos a que

$$LV(\theta) = \sum_{i=1}^{N} \sum_{j=1}^{J-1} y_j^i(\alpha_j^T x^i + \beta_j) - \sum_{i=1}^{N} \log\left(1 + \sum_{j=1}^{J-1} \exp(\alpha_j^T x^i + \beta_j)\right) \ .$$

Necesitamos ahora encontrar el valor $\widehat{\theta}$ que maximiza esta función. La resolución de este problema de optimización involucra la aplicación, nuevamente, de métodos iterativos. Por último, con los parámetros estimados, logramos definir estimadores para $p_j(x) = P(Y = E_j \mid X = x)$:

$$\hat{p_j}(x) := p_j(x;\widehat{\theta}) = \frac{\exp(\hat{\alpha}_j^T x + \hat{\beta}_j)}{1 + \sum_{k=1}^{J-1} \exp(\hat{\alpha}_k^T x + \hat{\beta}_k)} \ , \ j \in \{1, \ldots, J-1\}$$

$$\hat{p_0}(x) := p_0(x;\widehat{\theta}) = \frac{1}{1 + \sum_{k=1}^{J-1} \exp(\hat{\alpha}_k^T x + \hat{\beta}_k)}$$

En las mismas líneas que el caso binario, la regla de decisión quedará definida como $C : \mathbb{R}^d \to E$ tal que:

$$C(x) = \underset{j \in \{0,..,J-1\}}{arg\,max} \ \hat{p_j}(x)$$

Capítulo 5

Redes Neuronales Fully Connected

5.1. Fundamentos de las redes neuronales

En esta sección sentaremos las bases de la conformación de redes neuronales Fully Connected también llamadas perceptrones multicapa (*Multilayer Perceptrons*), siguiendo los trabajos de *Goodfellow, Bengio* y *Courville* en *Deep Learning*, *Ripley* en *Neural Networks and Related Methods for Classification* y *Bradley Efron and Trevor Hastie* en *Computer Age*; donde se pueden encontrar distintos abordajes de estos modelos.

Si bien en esta tesis estamos interesados en el problema de clasificar imágenes faciales en distintas emociones, haremos la presentación en un marco general. Fijemos entonces la notación con la que vamos a trabajar en este capítulo:

- (X, Y) es un vector aleatorio en $\mathbb{R}^d \times E$, donde X es el vector de **inputs**, atributos o variables predictoras, mientras que Y representa una variable categórica tomando valores en el conjunto de etiquetas $E = \{E_0, \ldots, E_{J-1}\}$.
- x denota un elemento en $\mathbb{R}^d$, con coordenadas $x = (x_1, \ldots, x_d)$.
- (x^i, y^i), $i \in \{1, \ldots, N\}$, son N observaciones (o realización) del vector (X, Y).

Recordemos que en el problema de imágenes, tenemos una matriz $x \in \mathbb{R}^{n \times n}$ con niveles de intensidad en cada píxel, es decir en cada celda. Para facilitar la lectura del manuscrito, en esta sección pensaremos a x como un vector concatenando todos sus elementos, diremos que $x \in \mathbb{R}^d$ con $d = n \times n$.

Cuando creamos oportuno, haremos alusión a este problema base aplicado para introducir diferentes conceptos.

5.1.1. Perceptrón y el modelo logístico

Retomemos por un momento el contexto de un modelo binario, por ejemplo digamos que lo único que nos interesa es clasificar si una imagen tiene expresión enojada o no luego $E = \{E_0, E_1\}$.

Comencemos el camino con las ideas del Capítulo 4, introduciendo un cambio de notación, con el propósito de respetar la tradición de cada cultura (estadística - computación). En primer lugar tomemos un predictor lineal $w^T x + b$, $w \in \mathbb{R}^d, b \in \mathbb{R}$. Esta simple expresión puede usarse para clasificar, es decir dividiendo el espacio de observaciones $\mathbb{R}^d$ con un hiperplano definido por $\{x : w^T x + b = 0\}$, de forma que dependiendo de qué lado caiga la observación será la etiqueta. A este modelo se lo conoce como *perceptrón*, fue desarrollado en 1958 por Frank Rosenblatt.

Siguiendo con el enfoque probabilístico, podemos componer el predictor lineal con la función sigmoidea $S(t) = \frac{1}{1+e^{-t}}$ y obtenemos la discriminación logística presentada en la Sección 4.1. En la literatura de redes neuronales este modelo se corresponde con una red neuronal simple FC conformada por *la capa de inputs* y *la capa de outputs*.

Dado un vector x, llamaremos al conjunto de sus coordenadas x_i la primera *capa* de la red, también denominada *capa de inputs*. Cada elemento de una capa, en este caso las coordenadas, se llamarán *neuronas*.

Por otra parte, la última capa de la red, será la *capa de outputs* que estará conformada por las neuronas necesarias para estimar las probabilidades de pertenencia a cada clase.

Recordando la Sección 4.1, en el caso binario basta con determinar la probabilidad de pertenecer a una de las clases (típicamente E_1). Tenemos entonces una única neurona en la capa de salida.

Notemos que esta regla particular consta de $d + 1$ parámetros. Podemos visualizarla en la siguiente imagen:

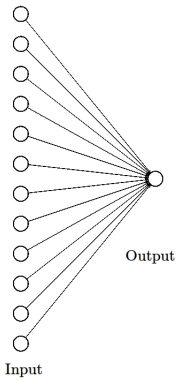


Figura 5.1: "Red input-output"

De esta manera, podemos pensar que esta red equivale a proponer, como asumimos en el modelo logístico, modelar $P(Y = E_1|X)$ con $p_1(X; (w, b))$ tal que:

$$p_1(x; (w, b)) = S(w^T x + b) = \frac{e^{w^T x+b}}{1 + e^{w^T x+b}} \tag{5.1}$$

5.1.2. Projection Pursuit Regression

Para ir construyendo un camino hacia redes más complejas, exploraremos otras formas de modelar una versión más general de la propuesta anterior.

Projection Pursuit Regression, más conocido como PPR, es un modelo que fue presentado en el paper *Projection Pursuit Regression (1981)* de los autores *Jerome H. Friedman y Werner Stuetzle* en el marco de la predicción, cuando la variable respuesta es continua. Su propósito fue encontrar un equilibrio entre los procedimientos no paramétricos (vecinos cercanos o promedios locales) y los procedimientos paramétricos (modelo lineal) para estimar la función de regresión.

Fue una propuesta sumamente innovadora que, a la luz del conocimiento actual, puede ser considerada una generalización de las redes neuronales conformadas por una capa oculta, concepto que introduciremos en breve.

En resumen, PPR propone incorporar a los clásicos modelos lineales de regresión para Y continua, lo siguiente:

- Incorporar proyecciones en distintas direcciones, incluyendo nuevos predictores lineales.
- Componer los predictores lineales con funciones suaves desconocidas y combinar aditivamente estas componentes.

De esta forma se consigue agregar otro tipo de funciones que el mismo procedimiento debería elegir y se permite también incluir interacciones entre las componentes del vector de variables predictoras.

Utilizaremos $w_k^{1T}x + b_k$, $k = 1, \ldots, m$ para construir m predictores lineales, con $w_k^1 \in \mathbb{R}^d, b_k \in \mathbb{R}$, que compondremos con funciones g_k desconocidas. Combinamos cada uno de estos términos aditivamente. Llegamos así a proponer el siguiente modelo para la $E(Y \mid X = x)$:

$$\sum_{k=1}^{m} g_k(w_k^{1T}x + b_k)$$

Es posible interpretar que, para evitar estimar las funciones g_k y acelerar un poco los tiempos de entrenamiento, las redes neuronales FC proponen, de alguna forma, parametrizar las funciones g_k.

Es decir, introduciendo unos segundos pesos $w_k^2 \in \mathbb{R}$, asumimos que $g_k(u) = w_k^2\sigma(u)$ donde $\sigma : \mathbb{R} \to \mathbb{R}$ es conocida como la *función de activación.* Sumaremos sobre k y agregamos un término adicional $b^2 \in \mathbb{R}$.

Si incorporamos esta última modificación podremos proponer un nuevo modelo para $E(Y \mid X = x)$:

$$\sum_{k=1}^{m} w_k^2\sigma(w_k^{1T}x + b_k) + b^2$$

En el presente contexto, considerando el camino que hemos recorrido, podemos reinterpretar esta propuesta y utilizarla para generalizar el modelo logístico en el problema de clasificación binario.

Formalmente, sea el vector $a = \big(\sigma(w_1^{1T}x + b_1), \ldots, \sigma(w_m^{1\,T}x + b_m)\big)$ que conforma lo que se conoce como la primera *capa intermedia u oculta* de la red y definimos para $k \in \{1, \ldots, m\}$:

- $a_k = \sigma(w_k^{1T}x + b_k)$, la k-ésima coordenada de a, representa la neurona k de la capa intermedia.
- $w_k^2 \in \mathbb{R}$ es el peso para la neurona k de la capa intermedia.

Finalmente definimos un sesgo $b^2 \in \mathbb{R}$. Si $\mathcal{W}$ denota el vector con todos los parámetros involucrados, podemos proponer el siguiente modelo para la $P(Y = E_1|X)$ con $p_1(X; \mathcal{W})$ tal que:

$$p_1(x; \mathcal{W}) = S\left(\sum_{k=1}^{m} w_k^2 a_k + b^2\right) \tag{5.2}$$

Podemos visualizar estas combinaciones en la siguiente imagen:

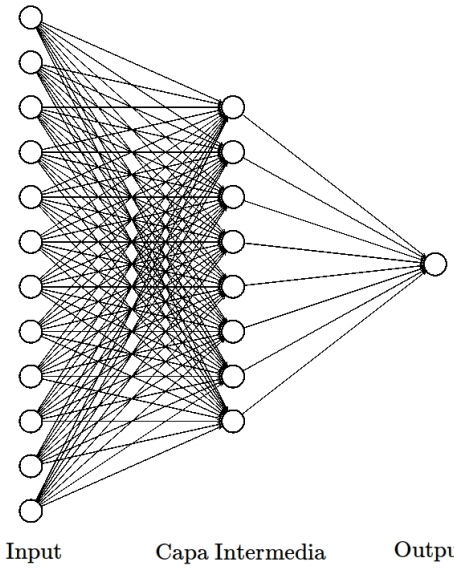


Figura 5.2: "Red con capa intermedia"

Notemos ahora que esta red propuesta con una capa intermedia se vuelve más abstracta, el modelo invita a determinar $d * m$ parámetros para las proyecciones iniciales w_k^1; m parámetros que hacen el papel de *intercept* b_k; m pesos w_k^2 y el coeficiente adicional $b^2 \in \mathbb{R}$. Necesitamos determinar $d*m+m+m+1$ parámetros en orden de tomar una decisión.

Intuitivamente le estamos otorgando más herramientas para obtener más detalles de la realidad que buscamos representar.

En este punto, con la inserción de capas ocultas a la red, es donde la cultura de la estadística y la computación empiezan a diferir. La estadística se preocupa por la *identificabilidad* de los parámetros, es decir le interesa que sean únicos, para luego poder definir nociones como intervalos de confianza y tests de hipótesis sobre ellos. En el mundo del aprendizaje automático esto pasa a un lugar secundario, el objetivo primario será *predecir* por lo que, en este contexto, no nos detendremos a considerar si se está proponiendo un modelo sobreparametrizado.

5.1.3. Estructura general

En esta sección complejizamos el modelo proponiendo agregar nuevas capas ocultas con diferente cantidad de neuronas para cada una de ellas. Denotaremos la *profundidad* de una red como la cantidad de capas ocultas a ser entrenadas.

Para unificar la notación, definimos a^0 a la capa de inputs y $a_j^0 = x_j$ siendo la j-ésima neurona de la capa inicial. Luego tendremos definida la siguiente forma recursiva para la salida de la neurona j de la capa intermedia a^l (también llamada activación):

$$a_j^l = \sigma^l \left(\sum_{k=1}^{m_{l-1}} w_{j,k}^l a_k^{l-1} + b_j^l \right) \tag{5.3}$$

donde:

- $j \in \{1, .., m_l\}, l =\in \{1, .., L\}$ siendo m_l la cantidad de neuronas en la capa l y L la última capa intermedia de la red.
- $w_{j,k}^l$ es el peso que se le otorga a la neurona k de la capa $l-1$ en la neurona j de la capa l.
- a_k^{l-1} la activación de la neurona k de la capa $l-1$.
- b_j^l es el sesgo para la neurona j de la capa l.
- σ^l es la función de activación de la capa l.

Finalmente, concatenamos con S para formar la capa de outputs y modelamos $P(Y = E_1|X)$ con $p_1(X; \mathcal{W})$ de forma que:

$$p_1(x; \mathcal{W}) = S\left({w^{L+1}}^T a^L + b^{L+1} \right) , \quad \text{con } w^{L+1} \in \mathbb{R}^{m_L},\, b^{L+1} \in \mathbb{R}, \tag{5.4}$$

donde, utilizamos $\mathcal{W}$ para denotar al vector con todos los parámetros involucrados (pesos y sesgos).

El hecho de que siempre una capa se alimenta de la anterior es lo que le da a este tipo de arquitecturas el carácter de ser un procedimiento *feed forward* siempre el flujo de información es hacia adelante. Son las más famosas pero cabe destacar que otras como *recurrent neural networks*, que están fuera de este estudio, permiten ciclos de información entre las capas.

En cuanto a la intuición para definir la cantidad de capas y neuronas que tendrá la red se recomienda siempre buscar bibliografía sobre arquitecturas ya implementadas según el problema que se quiera resolver. La arquitectura *ïdeal"* para resolver una tarea debe ser encontrada vía experimentación guiada por un monitoreo de cierta noción de error sobre un conjunto de datos de testeo.

5.1.4. Adaptación para resultados multinomiales

Con esto estamos en condiciones de levantar la restricción que le pusimos a nuestro problema original: no busquemos una respuesta binaria sino una multinomial con $J = 7$, queremos clasificar con cual de los 7 sentimientos (Enojo, Disgusto, Miedo, Felicidad, Tristeza, Sorpresa o Neutral) se corresponde la imagen. Para esto crearemos una arquitectura con la particularidad de que ahora la última capa no va a tener una neurona, va a tener J.

Por esto cambiaremos la función de activación de la última capa de la *Sigmoidea* a la función conocida como *Softmax*.

En la sección 4.2 cuando definimos el modelo multinomial dejamos afuera de la definición general a $p_0(x)$, ya que queda unívocamente determinada por $p_j(x)$, $j \in \{1, \ldots, J-1\}$. En este caso no repararemos en este hecho y definiremos la función Softmax como:

$$softmax : \mathbb{R}^J \to [0,1]^J \text{ donde}$$

$$softmax_j(z_0, .., z_{J-1}) = \frac{e^{z_j}}{\sum_{l=0}^{J-1} e^{z_l}} \qquad j \in \{0, ..., J-1\} \tag{5.5}$$

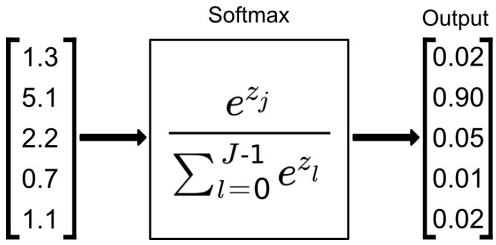


Figura 5.3: "Ejemplo Softmax"

La razón de usar esta función viene del efecto de usar como potencias los valores z_j y así exagerar las diferencias entre ellos. Esta función devolverá valores cercanos a 0 si z_j es significativamente menor que los demás y devolverá valores cercanos a 1 cuando sea el máximo (y la diferencia con los anteriores sea significativa). La suma de las probabilidades será 1.

Retomando la construcción de la arquitectura, usamos J neuronas para la última capa con la función de activación softmax y procedemos análogamente

a la expresión definida en (5.4), si se tienen L capas ocultas y definimos $w_j^{L+1} \in \mathbb{R}^{m_L}, b_j^{L+1} \in \mathbb{R}\, \forall j \in \{0, ..., J-1\}$, proponemos el siguiente modelo para la $P(Y = E_j \mid X)$ con $p_j(X; \mathcal{W})$ tal que:

$$p_j(x; \mathcal{W}) = softmax_j(w_0^{L+1T} a^L + b_0^{L+1}, ..., w_{J-1}^{L+1T} a^L + b_{J-1}^{L+1}) \qquad j \in \{0, ..., J-1\} \tag{5.6}$$

donde, nuevamente, utilizamos $\mathcal{W}$ para denotar al vector con todos los parámetros involucrados.

Por simplicidad dibujemos un esquema con una sola capa intermedia para darnos una idea visual de lo que está ocurriendo.

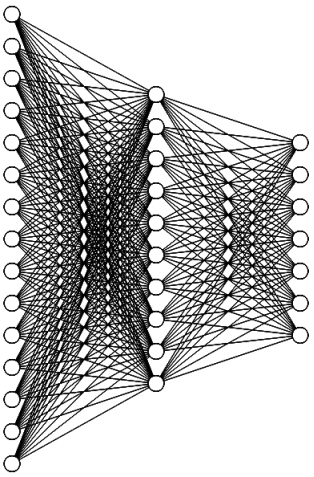


Figura 5.4: "Red multinomial-7"

5.2. Entrenamiento de la red.

En la sección anterior escribimos la definición del modelo de redes propuesto, ahora debemos indicar cómo elegir el vector $\mathcal{W}$ con todos los parámetros. En el Capítulo 4 mostramos un posible camino para seleccionarlos, en el caso de la discriminación logística, por el método de máxima verosimilitud.

En la literatura, se procura minimizar una función de pérdida. En el caso binario vamos entonces a minimizar la función

$$P(\mathcal{W}) = -\sum_{i=1}^{N} y^i \log\left(p_1(x^i; \mathcal{W})\right) + (1 - y^i) \log\left(1 - p_1(x^i; \mathcal{W})\right). \tag{5.7}$$

En el caso multiclase, retomando la notación propuesta, utilizamos la representación vectorial para las etiquetas. Tenemos así que la función de

pérdida a minimizar para elegir los parámetros del modelo está dada por

$$P(\mathcal{W}) = \sum_{i=1}^{N} P'(y^i, p(x^i; \mathcal{W})), \quad \text{con} \quad P'(y, p) := -\sum_{j=0}^{J-1} y_j \log(p_j) \tag{5.8}$$

donde $p(x^i; \mathcal{W}) = (p_0(x^i; \mathcal{W}), \ldots, p_{J-1}(x^i; \mathcal{W}))$.

Esta pérdida presentada, basada en la noción de verosimilitud, es la más usada en la práctica y se la conoce como *Cross entropy*.

El entrenamiento, elegir/aprender los parámetros, se reduce de nuevo a un problema de optimización, encontrar $\widehat{\mathcal{W}}$ tal que:

$$\widehat{\mathcal{W}} = \text{argmin}_{\mathcal{W}} P(\mathcal{W})$$

Es decir, idealmente buscamos $\widehat{\mathcal{W}}$ tal que $P(\widehat{\mathcal{W}}) \leq P(\mathcal{W})$ para todo $\mathcal{W}$. La tarea de optimizar esta expresión debe resolverse por métodos iterativos. En este estudio nos centraremos en el método más clásico revisado en la literatura para redes neuronales que se conoce como *descenso por el gradiente*. Veremos sus versiones *clásica* y *estocástica*.

Regularización

En este contexto, se pueden proponer distintas funciones de pérdida.

En algunos casos se puede agregar un término de regularización de los parámetros de forma que:

$$P(\mathcal{W}) = \sum_{i=1}^{N} P'(y^i, p(x^i; \mathcal{W})) + \lambda * J(\mathcal{W}_p)$$

donde W_p denota los parámetros involucrados que son pesos y no sesgos, $J(\mathcal{W}_p)$ es un término de regularización no-negativo, J una noción de norma, tamaño, de los parámetros y λ un hiperparámetro a definir.

Debemos observar que modificar la función de pérdida no es la única técnica que apunta a regularizar el modelo en redes neuronales; Otra opción es agregar capas intermedias que NO involucren la adhesión de parámetros a entrenar. Los ejemplos más clásicos son:

- Dropout: Las capas de dropout previenen el overfitting, reciben como input un valor Φ, fijado con anterioridad, que se interpreta como la probabilidad de que cada una de las neuronas de la capa inmediatamente anterior sean forzadas a valer 0, mientras que las que sobreviven

se multiplican por $1/(1+\Phi)$. De alguna forma las neuronas resultantes distintas de cero compensan las que se omiten.

- Batch Normalization: intenta reparametrizar los inputs de la capa inmediatamente anterior a su implementación. Dado un conjunto de activaciones le restaremos su media y la dividiremos por su desvío. Es de vital importancia en redes muy profundas para evitar valores extremos a los que las funciones de activación no sean sensibles.

En resumen, transforman el resultado de una capa, manteniendo sus dimensiones, antes de ser consumida por la siguiente. Para la cantidad y momento donde ubicar estas capas se recomienda,como siempre, leer bibliografía de acuerdo al problema que se busque resolver.

5.2.1. Descenso por el gradiente (GD)

Empecemos ahora a describir el método numérico propuesto para optimizar, Descenso por el gradiente (GD). En esencia busca valores donde se maximiza o minimiza una función f. Se trata de una algoritmo iterativo que computa repetidamente el gradiente de la función f (la dirección de mayor crecimiento de f) y se mueve un *paso* en la dirección opuesta. Si $f : \mathbb{R}^p \to \mathbb{R}$, dado un punto inicial v_0 y un paso η, definimos (v_t), siendo la sucesión generada como:

$$v_t = v_{t-1} - \eta \nabla f(v_{t-1}),$$

donde $\nabla f(v)$ denota el vector de derivadas parciales de la función f. Se puede demostrar que si la función f es convexa, la sucesión converge a su mínimo (Teorema de convergencia de Descenso por el gradiente). El *paso* η es conocido como *learning rate* y su elección no es para nada trivial.

En el presente contexto, actualizaremos los parámetros de forma que dada dado el paso t,

$$\mathcal{W}_{t+1} = \mathcal{W}_t - \eta \nabla P(\mathcal{W}_t) = \mathcal{W}_t - \eta \sum_{i=1}^{N} \nabla P(y^i, p(x^i; \mathcal{W}_t))$$

donde ∇ representa las derivadas parciales respecto a las coordenadas de $\mathcal{W}$.

Cabe notar que, en general, la función de pérdida no será convexa. El gradiente puede anularse en mínimos locales y en puntos de ensilladura. En la práctica se lo utiliza igual a pesar de esta observación y suele funcionar muy bien a los efectos predictivos para los cuales fue propuesto el método. Caracterizar cuáles son los parámetros que aprende es un campo de prematuro desarrollo.

En la práctica, se puede utilizar un *learning rate* que varíe a lo largo de las épocas para poder explorar nuevas regiones y no quedar estancado en puntos donde el gradiente se anula. También se recomienda considerar diferentes puntos iniciales v_0. Veremos algo relacionado a esto en la sección 6.3.

5.2.2. Descenso por el gradiente estocástico

En nuestro problema de interés, la función a minimizar, presentada en (5.8) consiste en una que suma de N términos, uno por cada dato del conjunto de entrenamiento.

Uno de los problemas que presenta el método anterior es el cálculo del gradiente para cada uno de los términos involucrados en la sumatoria. Para bases de datos muy grandes, hacer estos cálculos uno por uno se vuelve una tarea muy desafiante.

Una posible solución para bajar el tiempo de cómputo es introducir la noción de *mini batch* que será un subconjunto aleatorio de las observaciones significativamente más pequeño. Dado $M << N$ remuestraremos sin reposición nuestro conjunto de datos obteniendo (x^{i_m}, y^{i_m}) $m \in \{1, .., M\}$ y aproximaremos la dirección:

$$\frac{\sum_{i=1}^{N} \nabla P(y^i, p(x^i; \mathcal{W}))}{N} \approx \frac{\sum_{m=1}^{M} \nabla P(y^{i_m}, p(x^{i_m}; \mathcal{W}))}{M}$$

Podemos pensar que la suma de la derecha es una *esperanza*, que estamos aproximando por un promedio con valores elegidos al azar. Cuantos menos gradientes debamos calcular, menos cómputo requerirá el entrenamiento y por lo tanto más rápido será. Generalmente, los divisores que hacen al promedio suelen omitirse y simplemente se unen a la escala del learning rate η; haremos el abuso de notación de omitirlos. De esta manera, dado $\mathcal{W}_0$ inicial, al implementar el método del descenso por el gradiente estocástico (SGD), tenemos para el paso t:

$$\mathcal{W}_{t+1} = \mathcal{W}_t - \eta \sum_{m=1}^{M} \nabla P(y^{i_m}, p(x^{i_m}; \mathcal{W}))$$

Dicho de otra manera, iremos modificando los parámetros de forma que, para cierto M y η, se sobrescriban:

$$\mathcal{W} \longleftarrow \mathcal{W} - \eta \sum_{m=1}^{M} \nabla P(y^{i_m}, p(x^{i_m}; \mathcal{W}))$$

.

Definimos una época de entrenamiento como la declaración de que en ella todas las observaciones de entrenamiento fueron usadas en los cálculos de descenso por el gradiente irrespectivamente de cuantos pasos del algoritmo de optimización se han hecho (en el caso de usar mini-batch's).

5.2.3. Back propagation

Para terminar resta calcular $\nabla P(y, p(x; \mathcal{W}))$. Backpropagation es el algoritmo que efectivamente nos permitirá computar el gradiente de la función de pérdida de manera eficiente, permitiendo una implementación computacional exitosa del método.

La idea es calcular derivadas parciales de manera secuencial, comenzando por los últimos parámetros involucrados en la red, e invocando la *regla de la cadena multivariada generalizada*. Con esto se obtendrán fórmulas recursivas para las diferentes componentes del gradiente de la función de pérdida. Se puede ver su desarrollo en *Neural Networks and Deep Learning* de *Michael Nielsen*.

5.3. Predicciones

De esta forma, finalmente con los parámetros aprendidos $\hat{\mathcal{W}}$, si tenemos una red con una capa inputs con d neuronas, L capas intermedias y la capa de outputs con $J = 7$ *salidas*; clasificaremos la imagen según la neurona que haya dado el mayor valor de activación, es decir, mayor probabilidad de pertenecer a la clase, invocando así nuevamente el criterio de plug in del clasificador Bayes. Predecimos de la siguiente forma:

$$\hat{y}(x) = \underset{k \in \{0,...,J-1\}}{arg\,max} \; p_k(x; \hat{\mathcal{W}})$$

$$= \underset{k \in \{0,...,J-1\}}{arg\,max} \; softmax_{\,k} \left(\hat{w_0}^{L+1T} \hat{a^L} + \hat{b_0}^{L+1}, ..., \hat{w_{J-1}}^{L+1T} \hat{a^L} + \hat{b_{J-1}}^{L+1} \right)(x)$$

5.3.1. Elección función de activación

En general para problemas de clasificación en la capa de los outputs, la función de activación se adecua según el problema de clasificación a resolver

(binario ó multiclase). Sin embargo en las capas intermedias de la red tenemos más libertad. Algunas de las posibles elecciones son:

- Sigmoidea: *definida en la sección 4.1*
- Tanh: la función tangente hiperbólica (Tanh) se define de la siguiente forma,

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

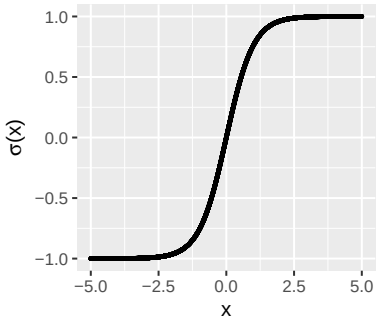


Figura 5.5: "Gráfico Tanh"

Esta función toma cualquier real y devuelve valores en el $[-1, 1]$. Cuanto mayor sea el input, más cercano a uno y cuanto menos, más cercano a -1.

Si observamos su forma, es muy parecida a la sigmoidea. Un problema que ambas comparten es que se saturan: las funciones son solo sensibles en un entorno alrededor de su punto medio (0 y 0.5 respectivamente). Esto hace que la tarea del algoritmo de optimización sea muy dificultosa: se generan gradientes pequeños que hacen que cuando se consideran muchas capas intermedias, el valor final del mismo se vuelva muy cercano a cero. Este problema se conoce como gradientes desvanecidos (vanishing gradients) y hace difícil encontrar la dirección por la que los parámetros deberían moverse para mejorar la función de costo.

- ReLU: la función ReLU (rectified linear activation function) se define de la siguiente forma,

$$\sigma(x) = max(0, x)$$

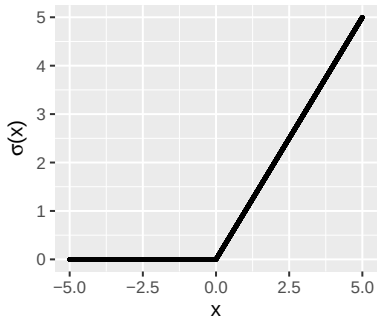


Figura 5.6: "Gráfico ReLU"

Es la de más frecuente uso. Algunas de las razones por la que lo es, son su fácil implementación y su efectividad a la hora de sobrellevar las limitaciones de las anteriores al ser menos susceptible a los gradientes desvanecidos.

En la práctica suele siempre usarse la misma función de activación en todas las capas ocultas. Tradicionalmente la sigmoidea era usada por defecto en la década de 1990, luego hasta mediados del 2010 la Tanh ocupó su lugar. En la actualidad las arquitecturas como la que vemos en este estudio, las FC y las convolucionales, hacen uso de la ReLU.

Una posible intuición del diseño de estas activaciones es buscar funciones que se las puedan considerar como una suavización derivable de una step function. Igualmente la decisión suele recaer en una solución pragmática.

5.4. Laboratorio Redes Neuronales

La idea de las secciones de laboratorio será disponibilizar código en Python para poder trabajar las técnicas vistas en el capítulo. Utilizaremos algunos conjuntos de datos artificiales para adentrarnos en distintos conceptos. La implementación puntual para el problema de reconocimiento de expresiones, mencionado en el Capítulo 3, la dejaremos para el cierre de la presente tesis.

Empezamos entonces mostrando, en un escenario controlado, la efectividad de armar estructuras más abstractas. Simulemos un problema binario, con dos clases: azules vs naranjas, nos ayudamos con la librería de Python *sklearn*:

```
from sklearn.datasets import make_circles
X, y = make_circles(n_samples=1000, noise=0.1, random_state=1)
```

Dividimos los datos en un conjunto de entrenamiento y uno de testeo manteniendo una relación de 80 %, 20 % respectivamente.

```
x_train, x_test, y_train, y_test = train_test_split(X, y,
train_size=0.8, stratify=y)
```

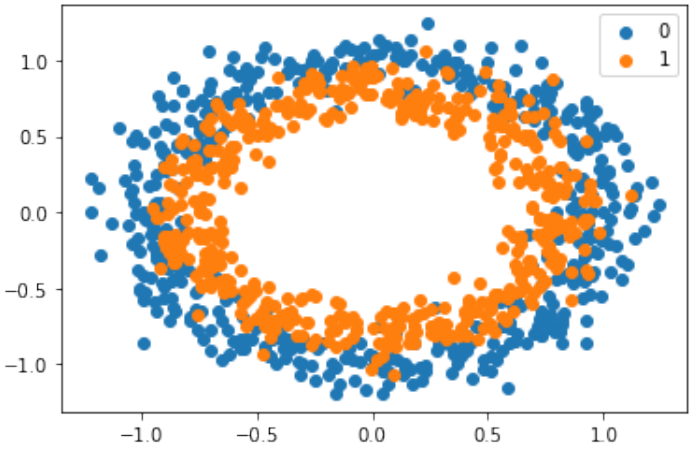


Figura 5.7: "Datos de entrenamiento"

Observemos, en primer lugar, que los puntos azules y naranjas se superponen en ciertos lugares del espacio por lo que es impracticable separarlos en su completitud, tendremos que tener cierta tolerancia a la hora de observar distintas métricas. En segundo lugar, es claro que las clases no responden, en ningún aspecto, a una separación lineal.

Veamos como responde a esto el modelo logístico (que observamos era equivalente a una red sin capas intermedias) y como una red con capas ocultas.

Comencemos con la discriminación logística, para su implementación usamos la función *LogisticRegression* de, nuevamente, la conocida librería de Python, *sklearn*.

```
from sklearn.linear_model import LogisticRegression

model_l = LogisticRegression(fit_intercept=True)
```

Recordemos que solo hay solo 3 parámetros involucrados, el peso de cada variable más un sesgo.

Entrenemos este modelo con el conjunto de entrenamiento y veamos como performa en el set de testeo:

```
model_l.fit(x_train, y_train)
```

La exactitud (en inglés *accuracy*), que se define como el promedio de clasificaciones correctas, obtenida en este experimento es de 0.47, veamos de forma gráfica este resultado:

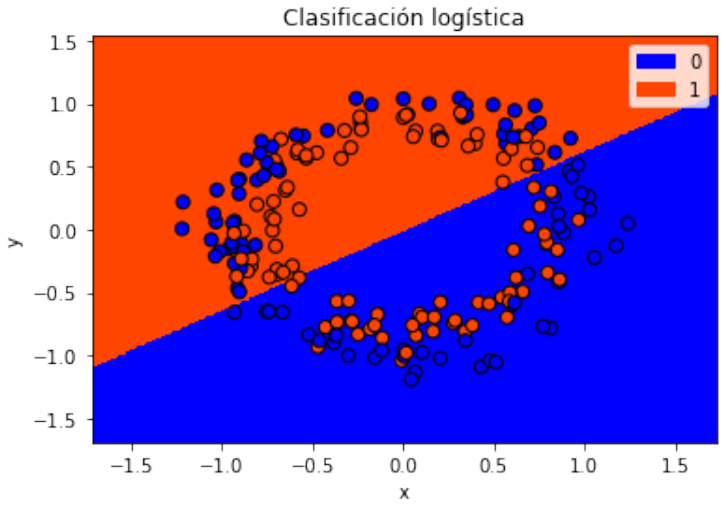


Figura 5.8: "Discriminación logística sobre los datos de testeo"'

Los puntos mantienen su etiqueta original y el fondo determina la parte del espacio que el modelo asigna a cada clase. Es claro que le erra a aproximadamente la mitad de los puntos.

Cabe aclarar que en la práctica, en caso de querer usar la regresión logística para resolver un problema se suelen agregar más variables explicativas que representen distintas interacciones entre las variables como por ejemplo $x_1^2, x_2^2, x_1 \times x_2$,etc. Este proceso manual se conoce como *Feature engineering* y es un gran punto a favor para las redes ya que veremos que, de alguna forma, lo tienen implícito en su formulación sin necesitar de un trabajo externo.

En estas líneas, implementamos una red con una capa intermedia de 32 neuronas.

Para programar las redes utilizaremos la librería open-source *tensorflow* que es desarrollada y mantenida por Google. Usaremos la aplicación *Keras* para conectarnos con *tensorflow* que está diseñada también por Google para ser modular, rápida e intuitiva.

Usaremos la función de activación RELU, el código y su sumario, que denota la cantidad de capas; neuronas y en general parámetros involucrados, se puede ver en las siguientes líneas:

```
model_fc = keras.Sequential([
      keras.layers.InputLayer(input_shape=(2)),
      keras.layers.Flatten(),
      keras.layers.Dense(32, activation='relu'),
      keras.layers.Dense(1, activation='sigmoid')
  ])
model_fc.summary()
Model: "sequential_17"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_17 (Flatten)         (None, 2)                 0
_________________________________________________________________
dense_28 (Dense)             (None, 32)                96
_________________________________________________________________
dense_29 (Dense)             (None, 1)                 33
=================================================================
Total params: 129
Trainable params: 129
Non-trainable params: 0
```

En este caso, tenemos que elegir 129 parámetros. Procedemos a entrenar de la misma forma que antes, en este caso debemos *compilar* y *fittear* el modelo. Elegimos como optimizador descenso por el gradiente estocástico con un learning rate de 0.01 y como función de pérdida la función cross-entropy binaria.

Obtenemos una exactitud del 0.82.

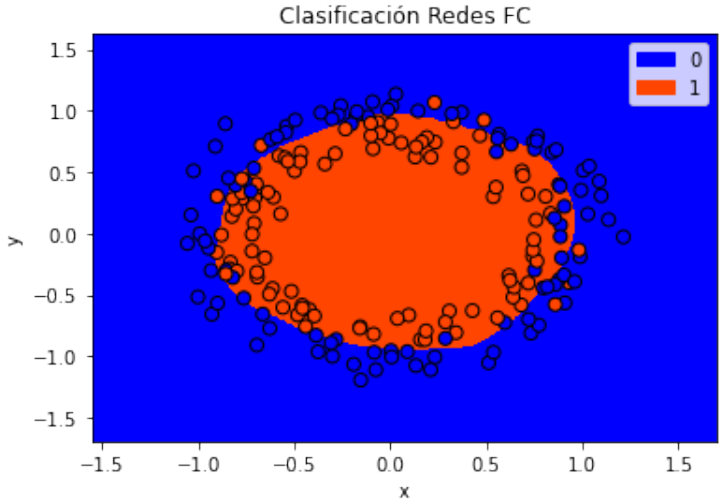


Figura 5.9: "Redes FC sobre los datos de testeo"

Vemos que sobre todo algunos puntos azules se nos escapan e ingresan a la zona naranja, pero en rasgos generales es capaz de reconocer la estructura de círculos mucho mejor.

Realizando una comparación entre ambos modelos utilizando la métrica de exactitud de clasificaciones correctas a través de validación cruzada utilizando 5 folds. Tenemos los siguientes resultados:

Técnica	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5
Discriminación Logística	0.50	0.48	0.43	0.46	0.47
Redes con capas intermedias	0.79	0.83	0.79	0.87	0.82

Teniendo en cuenta que es un modelo binario y decidirse por una clase o por otra tirando una moneda tiende a tener una exactitud de 0.5, el modelo logístico (red sin capas) hace un mal trabajo, mientras que con la introducción de una capa intermedia somos capaces de sortear el problema y poder realizar predicciones mucho más acertadas.

Capítulo 6

Redes Neuronales Convolucionales

6.1. Fundamentos de las redes neuronales convolucionales

Hasta ahora, en los métodos que vimos, tratamos las imágenes de entrada como un vector de tamaño d. En esta sección vamos a complejizar esa representación recordando que los datos son matrices ($d = n \times n$) y aplicando sobre ellas operaciones típicas. Indexaremos las distintas posiciones desde $(1, 1)$ a (n, n).

Seguiremos principalmente el trabajo de *Andrew Ng* en el curso *Convolutional Neural Networks* y nuevamente consultaremos el estudio de *Bradley Efron and Trevor Hastie* en *Computer Age*.

6.1.1. Filtros y feature maps

Las redes convolucionales, notadas frecuentemente por sus siglas en inglés *CNN*, observan la imagen x de a partes, es decir que la imagen se descompondrá en submatrices de $m \times m$. Cada submatriz es llamada campo local receptivo.

En primera instancia a cada submatriz la transformaremos *convolucionandola*, concepto que formalizaremos en breve, contra otra matriz w llamada *kernel* o filtro $\in \mathbb{R}^{m \times m}$, todas sus celdas estarán conformadas por pesos/parámetros que deberemos determinar en la etapa de entrenamiento de la red.

Además de fijar el tamaño m de cada filtro, de cada submatriz, debemos seleccionar un valor de *stride* s_c y un valor de padding p_c que se definen como

sigue.

Stride

El *stride* es el paso, denotado s_c, que daremos para diferenciar una submatriz de otra.

Este estará limitado por la posibilidad de dejar un espacio de $m \times m$ luego de haber realizado el paso.

Las elecciones más típicas de stride son $s_c = 1$ o $s_c = 2$.

Padding

El *padding* nace de las desventajas de esta clase de problemas:

- Las imágenes son matrices de dimensión finita, vamos a buscar resumirla por lo que luego en un número finito nos quedaremos sin matriz útil que analizar.
- Al tratarse de un procedimiento por submatrices con cierto valor de stride, inevitablemente los píxeles de las puntas, que podrían tener características esenciales, se recorren un número considerablemente menor que los del centro de la imagen.

La idea será elegir un valor p_c de modo que se le agrega, en el borde de la matriz de entrada, p_c filas y columnas con ceros. Por ejemplo con $p_c = 1$:

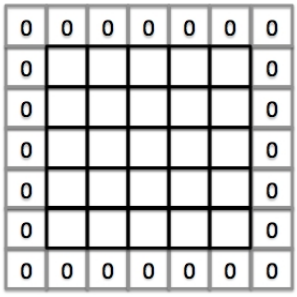


Figura 6.1: "Padding $p_c = 1$"

Esto evitará los dos problemas mencionados al principio ya que se visitarán más homogéneamente todas las celdas propias de la imagen e impediremos el colapso de la matriz.

Las elecciones más típicas de *padding* son:

- **Same**: Se elige p_c de forma que se agregan ceros suficientes tal que las matrices de input y output tengan las misma dimensión después de una

operación de convolución o pooling. Si el stride es 1 esto implica usar $p_c = \frac{f-1}{2}$. De acá se puede considerar que se genera la convención de que f sea impar.

- **Valid**: Se usará $p_c = 0$, en otras palabras no se utiliza relleno.

Con estos conceptos definidos describamos un ejemplo para fijar ideas, si tenemos una matriz de 6×6, "valid" padding $p_c = 0$, $s_c = 1$ y $m = 3$, observaremos la siguiente transformación:

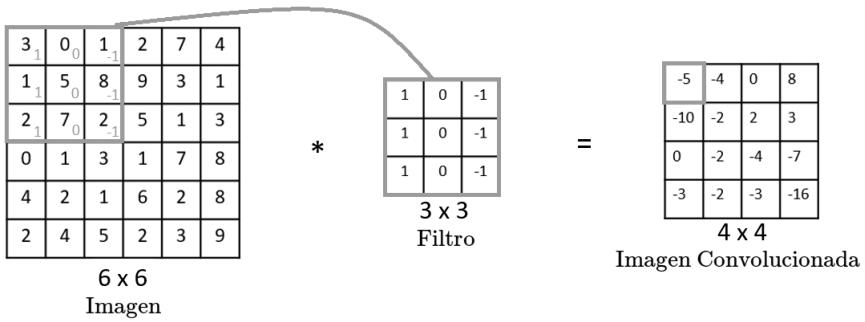


Figura 6.2: "Convolución"

En general una imagen x de tamaño $n \times n$, convolucionada por un filtro W de $m \times m$, padding p_c y stride s_c dará origen a una nueva matriz de dimensión de $(\frac{n-m+2p_c}{s_c} + 1) \times (\frac{n-m+2p_c}{s_c} + 1)$.

Si tomamos, en este caso, $s_c = 1$ y $p_c = 0$ podemos definir el operador de convolución $\otimes : \mathbb{R}^{n\times n} \times \mathbb{R}^{m\times m} \to \mathbb{R}^{(n-m+1)\times(n-m+1)}$ de la imagen $x \in \mathbb{R}^{n\times n}$ con el filtro $W \in \mathbb{R}^{m\times m}$:

$$(x \otimes w)_{i,j} = \sum_{k=1}^{m}\sum_{p=1}^{m} x_{i+k-1,j+p-1}W_{k,p} \qquad i,j \in \{1,...,n-m+1\}$$

A esta matriz, la imagen convolucionada, le sumaremos, como en el capítulo anterior, un sesgo b y una función no lineal σ. Así tendremos el concepto de *feature map* donde cada celda será una *neurona*. Luego cada componente, cada celda (i,j) resultante será:

$$\text{Feature map}_{i,j} = \sigma((x \otimes W)_{i,j} + b)$$

La intencionalidad del procedimiento estará en encontrar suficientes features maps de modo que seamos capaces de reconocer estructuras y características propias de la imagen. Así formaremos *capas convolucionales* que estarán compuestas por múltiples mapeos.

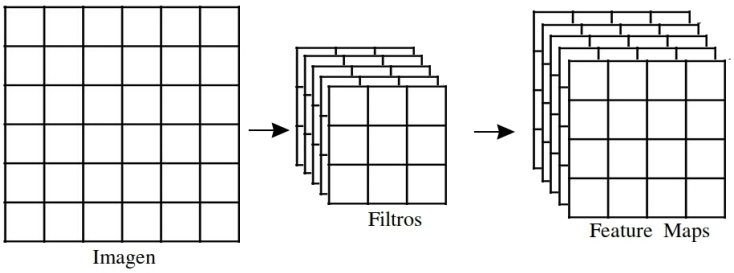


Figura 6.3: "Feature Maps"

Luego supongamos que queremos encontrar F *features maps* del input x, debemos entonces buscar F filtros $W_f^1 \in \mathbb{R}^{m\times m}, f : 1...F$ y sesgos $b_f^1 \in \mathbb{R}, f : 1...F$.

Pondremos como índice superior el número de capa convolucional en la que estamos y asumamos ahora $s_c = 1$, $p_c = 0$. Tomando σ como función activadora, la salida de cada uno de los primeros *feature maps*, $FM_f^1 \in \mathbb{R}^{(n-m+1)\times(n-m+1)}$, será :

$$FM_f^1 = \sigma(x \otimes W_f^1 + b_f^1) \tag{6.1}$$

6.1.2. Pooling

Otra herramienta particular que presentan las redes convolucionales son lo que se conoce como *capas de pooling* que en general son usadas inmediatamente después de una convolucional. En esencia toman cada *feature map* y preparan otro más "condensado" aún.

Estas capas dependen de un parámetro p y nuevamente de un valor s_o de stride y un p_o de padding. Cada submatriz de tamaño $p \times p$ será enviada a un solo valor. Por ejemplo tomando $p = 2, s_o = 2, p_o = 0$:

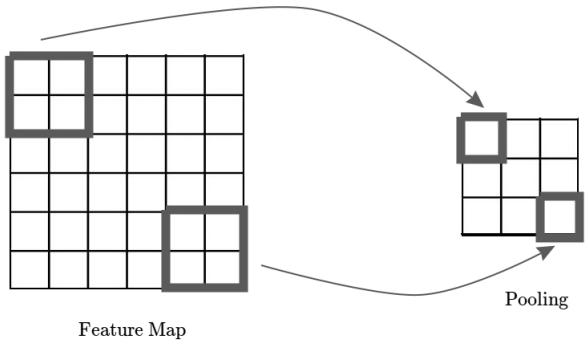


Figura 6.4: "Pooling Layer"

En general una imagen x de tamaño $n \times n$, al hacer un pooling de parámetros p, p_o y s_o dará origen a una nueva matriz de dimensión de $(\frac{n-p+2p_o}{s_o}+1) \times (\frac{n-p+2p_o}{s_o}+1)$.

Los operadores más usados son, nuevamente fijamos $s_o = 1$, $p_o = 0$ para simplificar la notación:

- **Average Pooling Layer**: Calcula el promedio de todos los elementos en cada campo local. Dado $FM_f^1 \in \mathbb{R}^{(n-m+1)\times(n-m+1)}$ la salida de la capa de pooling S_f^1 aplicada a FM_f^1 se define como:

$$(S_f^1)_{i,j} = \frac{1}{p^2}\sum_{h=0}^{p-1}\sum_{t=0}^{p-1}(FM_f^1)_{i+h,j+t} \qquad i,j \in \{1,...,(n-m+1)-p+1\} \tag{6.2}$$

- **Max Pooling Layer**: Calcula el máximo de todos los elementos en cada campo local. Dado $FM_f^1 \in \mathbb{R}^{(n-m+1)\times(n-m+1)}$ la salida de la capa de pooling S_f^1 aplicada a FM_f^1 se define como:

$$(S_f^1)_{i,j} = \max_{h,t\in\{0,...,p-1\}} (FM_f^1)_{i+h,j+t} \qquad i,j \in \{1,...,(n-m+1)-p+1\} \tag{6.3}$$

Los mapeos condensados son más robustos a cambios de posición del feature en la imagen (*invarianza por traslaciones locales*): pequeñas alteraciones en la misma imagen producirán *pooled feature maps* parecidos es decir con el feature en la misma locación.

6.1.3. Estructura red convolucional

Utilizando nuestro caso de estudio para reforzar el concepto, donde $X \in \mathbb{R}^{48\times48}$,$Y \in E$, estamos en condiciones de dar una propuesta de una posible red convolucional. Vamos a buscar una estructura que responda a la siguiente imagen (los tamaños en la imagen son meramente ilustrativos):

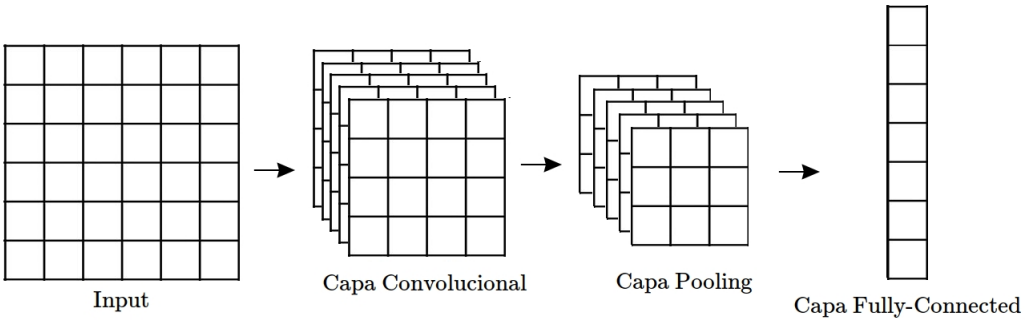


Figura 6.5: "Red Neuronal Convolucional"

Comenzamos con una realización de X, una imagen de 48×48 neuronas iniciales, le sigue una *capa convolucional* con 5 filtros de 3×3 ($s_c = 1, p_c = 0$) produciendo 5 *feature maps*. A continuación una *capa max pooling* de 2×2 a ($s_o = 2, p_o = 0$).Contamos, a este punto con 5 mapeos, también llamados 5 canales, de tamaño 23×23 neuronas.

Por último las redes convolucionales harán uso también de capas FC como las que vimos en el capítulo anterior para obtener la predicción final: se conecta cada una de las $5\times23\times23$ neuronas a cada una de las 7 finales que luego interpretaremos como probabilidades en favor o no de que la imagen representa a esa expresión, esto lo haremos seleccionando la función softmax.

Concatenamos las matrices resultantes de forma que nos quede un vector $f \in \mathbb{R}^{(5\times23\times23)\times1}$. Introduciendo nuevamente unos pesos $w_j \in \mathbb{R}^{(5\times23\times23)\times1}$ y sesgos $b_j \in \mathbb{R}\, \forall j \in \{0, ..., J-1\}$ vamos a proponer el siguiente modelo para la $P(Y = E_j \mid X)$ usando $p_j(X; \mathcal{W})$ tal que:

$$p_j(x; \mathcal{W}) = softmax_i(w_0^T f + b_0, ..., w_{J-1}^T f + b_{J-1}) \qquad j \in \{0, ..., J-1\} \quad (6.4)$$

donde nuevamente $\mathcal{W}$ es el vector con todos los parámetros involucrados (cada elemento de las matrices de filtros y los sesgos).

Estructura red convolucional generalizada

De la misma forma como en las redes FC agregamos tantas capas ocultas como se deseará, ahora este papel lo jugarán las capas convolucionales y las de pooling pudiendo agregar tantas como se quiera.

Supongamos que agregamos L capas ocultas, debemos concatenarlas como en 6.1, 6.2 y 6.3. Si llamamos f al vector que concatena los F_L feature maps resultantes luego de la última capa intermedia, el modelo que proponemos es, sean $w_j \in \mathbb{R}^{dim(f)}, b_j \in \mathbb{R}\ \forall j \in \{0, ..., J-1\}$, luego procedemos como en 6.4 modelando la $P(Y = E_j \mid X)$ con $p_j(X; \mathcal{W})$:

$$p_j(x; \mathcal{W}) = softmax_i(w_0^{L+1} f + b_0, ..., w_{J-1}^{L+1} f + b_{J-1}) \qquad j \in \{0, ..., J-1\} \tag{6.5}$$

donde $\mathcal{W}$ es el vector con todos los parámetros involucrados (cada elemento de las matrices de filtros, pesos y sesgos de las capas pertinentes).

Una observación válida es que en este estudio estamos dando una introducción a las redes convolucionales como las que contienen capas que operan estrictamente a nivel de matrices, en la literatura suelen encontrarse estructuras híbridas que luego de concatenar las neuronas en el vector f estamos en el escenario de inputs del capítulo anterior y se pueden concatenar algunas capas extras FC.

6.1.4. Nociones de entrenamiento

Nuevamente el siguiente paso es encontrar las estimaciones de los parámetros involucrados. El entrenamiento lo realizaremos igual que en el capítulo anterior, asumimos las mismas hipótesis para la pérdida P y utilizaremos la misma técnica de back-propagation que antes: por ejemplo en el caso base descrito nos interesará calcular las derivadas de P respecto de los pesos w y los sesgos b presentes en todos los filtros involucrados y la concatenación final.

Por último, iremos actualizando los parámetros, nuevamente, por algún método de optimización como descenso por el gradiente clásico o estocástico.

6.1.5. Predicciones

De esta forma con los parámetros aprendidos $\hat{\mathcal{W}}$, podremos clasificar la imagen según la neurona que haya dado el mayor valor de activación i.e. mayor probabilidad de pertenecer a la clase, invocando así nuevamente el criterio de plug in del clasificador Bayes.

$$\hat{y}(x) = \underset{k \in \{0,...,J-1\}}{arg\,max}\ p_k(x; \hat{\mathcal{W}})$$

$$= \underset{k\in\{0,...,J-1\}}{arg\,max} \; softmax_k(\hat{w_0}^{L+1}\hat{f} + \hat{b}_0, ..., \hat{w_{J-1}}^{L+1} + \hat{b}_{J-1}\hat{f})(x)$$

6.2. Ventajas y propiedades de las arquitecturas convolucionales

Numeraremos algunas posibles propiedades y ventajas de la utilización de la estructura de las redes convolucionales.:

- **Campos Locales receptivos (local receptive fields)**:

  La decisión de que cada nueva neurona se alimente de solo una porción de la imagen, el campo local receptivo, hace que dicha neurona se lleve la tarea directa de aprender patrones visuales dentro de esa región específica. Al contrario de las FC que todas se alimentan de todas las anteriores, tiene la propiedad de imitar lo que verdaderamente ocurre cuando el humano analiza una imagen, donde cada parte tiene su propia interpretación. Este fenómeno también se conoce como *sparcity of connections* (escasez de conexiones).

- **Compartir parámetros (parameter sharing)**:

  La aplicación sistemática de filtros a través de los distintos campos de una matriz es una idea muy poderosa, le da la oportunidad a la red de descubrir un mismo *feature/patrón* en cualquier lado de la imagen. El fenómeno se conoce como *parameter sharing* y también, al reducir drásticamente los parámetros en comparación a la redes FC, mejora considerablemente los tiempos de entrenamiento.

- **Reducción espacial (Spatial resolution reduction)**:

  La información que importa a la hora de pensar en la posición de un patrón es relativa a la posición de otros dentro del mismo feature map y no a su locación exacta en ese mapeo, en este último caso nuestras predicciones podrían empezar a parecerse demasiado a nuestros datos y no ser capaces de generalizar correctamente. Las CNN salvan este problema aplicando métodos de sub-muestreo como las operaciones de pooling: estas capas resumen cada campo local receptivo fijo en un único valor.

- **Invariancia por traslaciones (translation invariance)**:

  Esta última propiedad que mencionaremos se puede decir que es la culminación de la unión de las anteriores tres. La invariancia por traslaciones se puede definir, en este caso, como la habilidad de un algoritmo de

predecir el mismo output para un input y el mismo transformado por una operación T de traslación (rotaciones, desplazamientos): si corremos un par de píxeles el input lograremos la misma salida. Se trata de un procedimiento robusto en este sentido, en esencia ignorará cambios en la posición del patrón a identificar.

6.3. Laboratorio CNN

Al igual que en el laboratorio de redes neuronales buscaremos disponibilizar código en Python para poder trabajar las técnicas vistas en el capítulo utilizando *tensorflow* y *Keras*.

Empezamos nuevamente con un ejemplo de datos artificiales para mostrar, en un escenario controlado, la efectividad de armar redes convolucionales. Como antes nos armamos un ejemplo donde no se cumpliera el supuesto de linealidad, ahora armemos uno donde ganar la estructura de datos ordenados en una matriz sea importante. Por ejemplo pensemos en el ejemplo de reconocer números escritos a mano.

La base de datos MNIST de dígitos manuscritos tiene un conjunto de entrenamiento de 60 000 ejemplos y un conjunto de prueba de 10 000 ejemplos. Los dígitos tienen un tamaño normalizado y están centrados en una imagen de tamaño fijo de dimensión 28×28.

Tensorflow cuenta con su propio conjunto de datos listos para usar y MNIST es uno de ellos, podemos traernos las imágenes con el siguiente código:

```
import tensorflow_datasets as tfds
#reading the data
(ds_train, ds_test), ds_info = tfds.load(
    "mnist",
    split=['train', 'test'],
    shuffle_files=True,
    as_supervised=True,
    batch_size = -1,
    with_info=True,)
```

Veamos algunas de las imágenes que estamos usando:

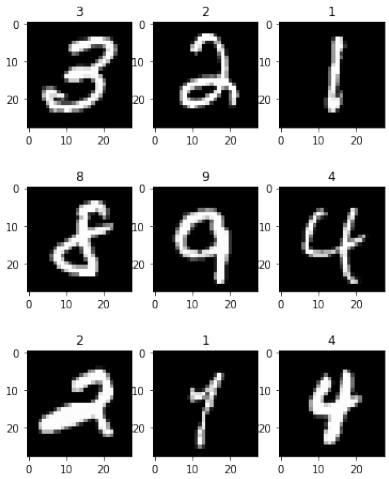


Figura 6.6: "MINST"

Al estar tratando con imágenes, es una buena práctica comenzar cualquier red utilizando una capa de normalización: se dividen todos los inputs por 255 que es el máximo valor de intensidad posible de un píxel, logrando así que todos los valores de entrada estén entre 0 y 1. Una posible intuición indica que esto propicia un buen escenario para el uso de las funciones de activación descriptas.

Definamos entonces para la tarea una red FC de 4 capas y veamos un sumario de cómo están conformadas y cuántos parámetros involucran:

```
model = tf.keras.Sequential([
    keras.layers.InputLayer(input_shape=(28,28, 1)),
    keras.layers.Lambda(lambda x: x/255.0),
    keras.layers.Flatten(),
    keras.layers.Dense(190, activation = "relu"),
    keras.layers.Dense(128, activation = "relu"),
    keras.layers.Dense(64, activation = "relu"),
    keras.layers.Dense(64, activation = "relu"),
    keras.layers.Dense(10, activation = "softmax")
])
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lambda (Lambda)              (None, 28, 28, 1)         0
```

```
_________________________________________________________________
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
dense (Dense)                (None, 190)               149150
_________________________________________________________________
dense_1 (Dense)              (None, 128)               24448
_________________________________________________________________
dense_2 (Dense)              (None, 64)                8256
_________________________________________________________________
dense_3 (Dense)              (None, 64)                4160
_________________________________________________________________
dense_4 (Dense)              (None, 10)                650
=================================================================
Total params: 186,664
Trainable params: 186,664
Non-trainable params: 0
```

Tenemos una red del tipo perceptron multi-capa de 4 capas ocultas con 190,128,64 y 64 neuronas respectivamente. Esto involucra la elección de 186664 parámetros.

Entrenamos la red con el set de entrenamiento por defecto de MINST y observamos su desempeño con el conjunto de testeo también por defecto, correspondiente a 10000 imágenes de números manuscritos.

Nuevamente debemos *compilar* y *fittear* el modelo.

Como ya adelantamos en el capítulo anterior, elegimos como optimizador una adaptación del descenso por el gradiente estocástico llamado *adam optimizer*, básicamente levanta la hipótesis de que tenemos un valor de learning rate η único para todos los parámetros, para todas las épocas de entrenamiento. En esencia se introducen los siguientes cambios:

- Cada parámetro involucrado tendrá su propio η.
- Se seleccionan learning rates iniciales y se irán actualizando según el primer y segundo momento del gradiente de cada parámetro.

Como función de pérdida la función sparse-cross-entropy que es la cross-entropy definida en 5.8 pero permite que las etiquetas entren como enteros (y no es necesario implementar previamente la codificación one-hot).

```
model.compile(optimizer = "adam",
    loss = tf.losses.sparse_categorical_crossentropy,
    metrics = ['accuracy'])
```

```
hist = model.fit(x = train_data, y = train_labs,
    batch_size = 32,
    epochs= 10,
    verbose = 1,
    validation_data = (val_data,val_labs))
```

Con esto podemos ver el desempeño en los datos de testeo. Utilizaremos la función *metrics.classification_report* de la librería *sklearn* sobre los datos de testeo para obtener un set de métricas resumen, notamos *support* de la clase i a la cantidad de imágenes de etiqueta verdadera i del set de testeo.

Las medidas se definen como lo siguiente:

- Precision: la definimos como a la exactitud, es la cantidad de clasificaciones correctas para cada una de las clases. Se define como, dada la etiqueta i:

$$\frac{\#\text{ clasificaciones correctas iguales i}}{\#\text{ clasificaciones iguales i}}$$

- Recall: se la conoce como sensibilidad en español; dada una clase es la proporción de etiquetas que fueron correctamente identificadas. Se define como, dada la etiqueta i:

$$\frac{\#\text{ clasificaciones correctas del support de i}}{\text{support de i}}$$

- F1 Score: es una combinación de las anteriores dos de forma que se mantiene el valor óptimo en 1, cuando ambas métricas mejoran, y el peor en 0 . Se define como:

$$\frac{2\ (\text{Recall} \times \text{Precision})}{\text{Recall} + \text{Precision}}$$

En función de esto encontramos los siguientes resultados:

```
y_pred=modelo_fc.predict(test_data).argmax(axis=1)
print(metrics.classification_report(test_labs, y_pred))
```

	precision	recall	f1-score	support
0	0.98	0.99	0.98	980
1	0.98	0.99	0.99	1135
2	0.98	0.97	0.98	1032

3	0.98	0.96	0.97	1010
4	0.96	0.98	0.97	982
5	0.97	0.97	0.97	892
6	0.98	0.96	0.97	958
7	0.97	0.99	0.98	1028
8	0.98	0.95	0.96	974
9	0.95	0.96	0.96	1009
accuracy			0.97	10000
macro avg	0.97	0.97	0.97	10000
weighted avg	0.97	0.97	0.97	10000

Además veamos gráficamente la evolución del algoritmo a lo largo de las épocas:

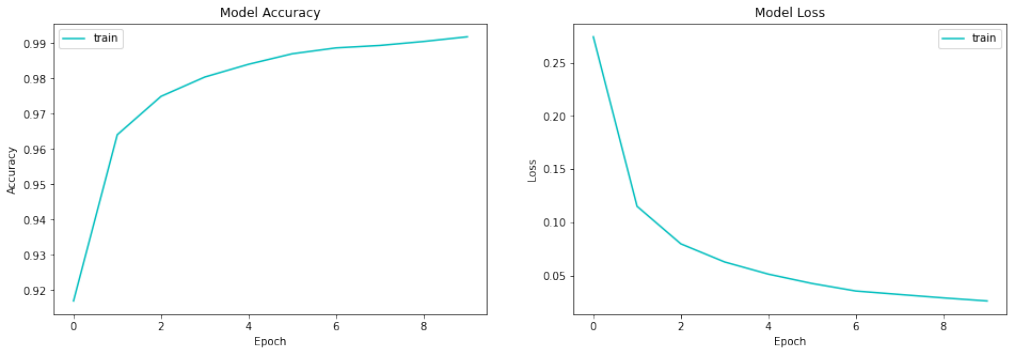


Figura 6.7: "Convergencia redes FC"

Recordemos que una época no significa un paso en el algoritmo de GD. Al estar usando una versión del SGD, para la primera época, por ejemplo, ya observamos un accuracy superior a 0.9 al ya estar implícitos varios pasos de actualización de los parámetros. Esto ocurre en función de haber tomado mini-batch de, en este caso, 32 observaciones (el valor por default de la herramienta).

Por otro lado, definimos una CNN también con 4 capas convolucionales seguidas cada una por una capa de Pooling. Veamos su implementación y sumario:

```
model = models.Sequential([
  keras.layers.InputLayer(input_shape=(28,28, 1)),
  keras.layers.Lambda(lambda x: x/255.0),
```

```
  keras.layers.Conv2D(256, (3,3), activation='relu'),
  keras.layers.MaxPooling2D(2,2),
  keras.layers.Conv2D(128, (2,2), activation='relu'),
  keras.layers.MaxPooling2D(2,2),
  keras.layers.Conv2D(64, (2,2), activation='relu',padding="same"),
  keras.layers.MaxPooling2D(2,2),
  keras.layers.Conv2D(64, (2,2), activation='relu',padding="same"),
  keras.layers.MaxPooling2D(2,2),
  keras.layers.Flatten(),
  keras.layers.Dense(10, activation='softmax'),
])
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lambda_1 (Lambda)            (None, 28, 28, 1)         0
_________________________________________________________________
conv2d (Conv2D)              (None, 26, 26, 256)       2560
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 13, 13, 256)       0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 12, 12, 128)       131200
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 6, 6, 128)         0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 6, 6, 64)          32832
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 3, 3, 64)          0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 3, 3, 64)          16448
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 1, 1, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 64)                0
_________________________________________________________________
dense_5 (Dense)              (None, 10)                650
=================================================================
Total params: 183,690
Trainable params: 183,690
```

```
Non-trainable params: 0
```

Tenemos una red del tipo convolucional, todas las capas convolucionales, estarán seguidas por una capa de pooling de stride y padding 2. La primera capa utilizará filtros de 3×3 para armar 256 canales. Las siguientes usarán filtros de tamaño 2×2 y formarán 128,64,64 canales respectivamente. Todo esto involucra la elección de 183690 parámetros.

Al testearla sobre el conjunto de prueba obtenemos los siguientes resultados:

```
y_pred=modelo_cnn.predict(test_data).argmax(axis=1)
print(metrics.classification_report(test_labs, y_pred))
```

	precision	recall	f1-score	support
0	0.97	1.00	0.98	980
1	0.97	1.00	0.99	1135
2	0.99	0.97	0.98	1032
3	1.00	0.96	0.98	1010
4	0.98	0.99	0.99	982
5	0.98	0.99	0.98	892
6	1.00	0.98	0.99	958
7	0.97	0.99	0.98	1028
8	0.99	0.99	0.99	974
9	0.99	0.97	0.98	1009
accuracy			0.98	10000
macro avg	0.98	0.98	0.98	10000
weighted avg	0.98	0.98	0.98	10000

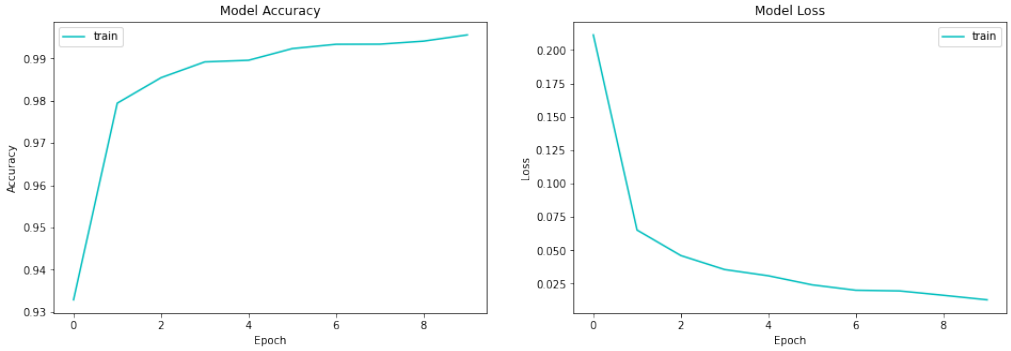


Figura 6.8: "Convergencia redes CNN"

En términos de cantidad de parámetros ambas redes deben elegir aproximadamente la misma cantidad. Las CNN parecen converger más rápido (menos épocas) que las FC. La exactitud en ambos casos no varía mucho, esto es porque la tarea es bastante guiada y tenemos números centrados por lo que esperamos que cada atributo se corresponda con un mismo píxel.

Sin embargo, una de las propiedades que proporcionan las redes convolucionales es la inserción de invarianza por pequeñas traslaciones como discutimos a lo largo del capítulo. Para proponer un ejemplo aplicado de esto vamos a traernos un conjunto de datos en el cual los números estarán trasladados aleatoriamente 3 píxeles a la derecha o a la izquierda y lo mismo en sentido hacia arriba o hacia abajo. Nuevamente hacemos uso de TFDS y nos traemos el set de datos de prueba del dataset *mnist_corrupted/translate.*

Ahora las imágenes se ven como las siguientes:

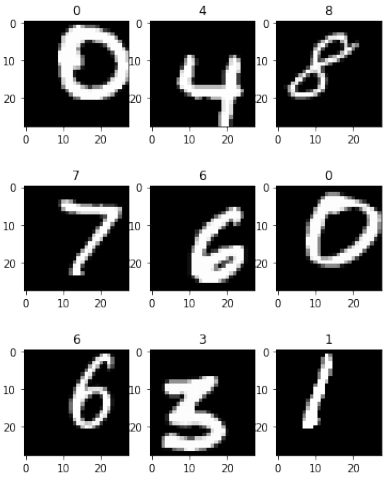


Figura 6.9: "MINST con corrupciones por traslaciones"

Usando los mismos modelos entrenados con imágenes centradas veamos su capacidad de ser robustos a estas traslación testeando en las 10000 imágenes de testeos corruptas. Obtenemos los siguientes resultados:

Resultados FC

```
y_pred=modelo_fc.predict(test_data_cor).argmax(axis=1)
print(metrics.classification_report(test_labs_cor, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.22      0.04      0.06       980
```

1	0.42	0.15	0.22	1135
2	0.30	0.45	0.36	1032
3	0.33	0.40	0.36	1010
4	0.44	0.26	0.33	982
5	0.31	0.52	0.39	892
6	0.32	0.43	0.36	958
7	0.32	0.46	0.38	1028
8	0.17	0.24	0.20	974
9	0.23	0.11	0.15	1009
accuracy			0.30	10000
macro avg	0.31	0.30	0.28	10000
weighted avg	0.31	0.30	0.28	10000

Resultados CNN

```
y_pred=modelo_cnn.predict(test_data_cor).argmax(axis=1)
print(metrics.classification_report(test_labs_cor, y_pred))
```

	precision	recall	f1-score	support
0	0.61	0.80	0.69	980
1	0.53	0.85	0.65	1135
2	0.87	0.60	0.71	1032
3	0.98	0.64	0.78	1010
4	0.86	0.76	0.81	982
5	0.81	0.84	0.83	892
6	0.67	0.59	0.63	958
7	0.64	0.84	0.72	1028
8	0.76	0.66	0.71	974
9	0.77	0.51	0.61	1009
accuracy			0.71	10000
macro avg	0.75	0.71	0.71	10000
weighted avg	0.75	0.71	0.71	10000

Recordemos que en este caso 0.1 es la exactitud de elegir al azar la etiqueta, las FC alcanzan una métrica de 0.3 mientras que las convolucionales 0.7. Se destaca claramente la adaptabilidad de estas últimas sobre estas introducciones de ruido sobre los datos. Una posible intuición nos indica que las capas convolucionales y, sobre todo, las de pooling actúan como capas

resumen de la imagen lo que provoca que la red tenga más robustez a la hora de observar datos con este tipo de perturbaciones.

Capítulo 7

Discusión: Calibración de los modelos

Hasta ahora hemos dado un marco descriptivo teórico de construcción de los modelos sin detenernos a pensar en las posibles hipótesis que deberíamos tener en cuenta para poder decir que los resultados serán óptimos. A la hora de evaluar su desempeño podemos plantear dos grandes objetivos que pueden ser independientes entre sí:

- Discriminar: es decir clasificar correctamente la mayor cantidad de veces, en otras palabras medir que tan bueno es mi algoritmo separando las clases.
- Calibrar: es la capacidad de interpretar correctamente los outputs de la capa de salida como probabilidades. Formalmente si $\hat{Y}$ es una predicción de la variable aleatoria Y y $\hat{P}$ es la confianza de predicción asociada, es decir la probabilidad asociada a la etiqueta predicha i.e. el máximo de la capa de salida, un modelo está perfectamente calibrado si $P(\hat{Y} = Y|\hat{P} = p) = p$.

Como mencionamos, discriminar/predecir es el objetivo primario en el desarrollo de las arquitecturas asociadas al aprendizaje automático, por lo que ese fue el foco de su evolución. Sin embargo, en estudios más recientes, se están proponiendo varios escenarios donde a causa de este objetivo los modelos podrían haber empeorado en otros aspectos como la calibración.

En las líneas de este estudio, la calibración nace de la necesidad de no solo modelar que el máximo del vector de outputs sea coincidente con las etiquetas si no, poder interpretar al vector como efectivamente probabilidades. Una posible motivación a esto podría ser que queremos remover elementos que no pertenezcan a ninguna de las clases que el problema propone, por lo que

querríamos descartar los casos donde el máximo de la salida sea muy bajo. Queremos usar el output del modelo como medida de incertidumbre de la clasificación realizada.

Por la construcción propuesta de las redes, si entrenamos el modelo para optimizar la pérdida Cross Entropy, luego esperamos que las salidas del modelo sean las probabilidades a posteriori, es decir $P(Y = E_j|X = x)$.

Sin embargo, la sobre-parametrización de estos modelos puede provocar que el algoritmo overfittee (se parezca demasiado a los datos) y se obtengan resultados subóptimos.

En el paper de los autores *Guo, Chuan Pleiss, Geoff Sun, Yu Weinberger, Kilian. (2017) llamado On Calibration of Modern Neural Networks.* muestran en un ejemplo aplicado como puede ocurrir esto. Para poder medir que tan calibrado está un modelo se proponen las siguientes definiciones.

Dado un conjunto de testeo (x^i, y^i) con $i \in \{1, .., N_T\}$ lo agrupamos en M intervalos, generando los conjuntos $B_m, m \in (1, ..., M)$ de acuerdo a si la confianza de su predicción cae en el intervalo $(\frac{m-1}{M}, \frac{m}{M}]$. Con esto definimos lo siguiente:

- $acc(Bm) = \frac{1}{|B_m|} \sum_{i \in B_m} 1_{\hat{y}_i = y_i}$ donde $\hat{y}_i$ es la etiqueta predicha por el modelo.

- $conf(Bm) = \frac{1}{|B_m|} \sum_{i \in B_m} \hat{q}_i$ donde $\hat{q}_i$ es la confianza de predicción para la observación i.

- *Reliability Diagrams* : son una representación visual de la calibración de un modelo. Estos diagramas grafican la exactitud observada $(acc(B_m))$ en función de la confianza $(conf(B_m))$. Se puede ver un ejemplo en la figura 7.1. Si el modelo está perfectamente calibrado esperaríamos ver una diagonal perfecta.

Midiéndose por esto muestran sobre un conocido dataset *CIFAR-100*, que cuenta con imágenes clasificadas en 100 categorías, el desempeño de dos arquitecturas: letnet 1998 (una red convolución de profundidad 5 propuesta en *Backpropagation applied to handwritten zip code recognition.* por Yann LeCun) vs Resnet 2016 (una red residual de profundidad 100 presentada en *Deep Residual Learning for Image Recognition* por Kaiming He).

Muestran que en este escenario la Resnet logra un error mucho menor que la Letnet pero en términos de calibración empeora.

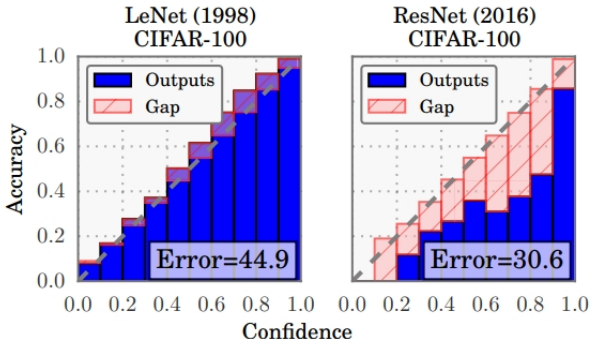


Figura 7.1: "Reliability diagrams. *On Calibration of Modern Neural Networks.*"

Entre sus comentarios mencionan que al aplicar capas de Batch Normalization y entrenar con menos penalización de los parámetros evidenciaron tener un efecto negativo sobre la calibración. Estos últimos son herramientas populares de las redes más modernas, como lo es la Resnet.

Una posible forma de mejorar la calibración del modelo para el problema de clasificación multiclase se conoce como *Temperature Scaling* que se define como, dado $T \in \mathbb{R}, T > 0$ sea z^i el vector resultante de la capa de salida antes de aplicar Softmax para el input x^i, definimos la nueva confianza en la predicción q^i como:

$$\hat{q^i} = \max_{k \in \{0,..,J-1\}} softmax_{\,k}\left(\frac{z^i}{T}\right)$$

El parámetro T se lo conoce como temperatura y suaviza el output de la softmax cuando $T > 1$. Como T está fijo, se lo elige optimizando la log verosimilitud, no cambia el máximo de la salida por lo que en términos de exactitud el modelo se ve inalterado.

En resumen la implementación de técnicas modernas para las cuales tenemos evidencia que podrían empeorar la calibración de los métodos no es necesariamente incorrecta. Debemos solo tener en claro el objetivo que queremos perseguir, en función de esto elegir una métrica que lo represente y considerar que la posible extensión a otro tipo de interpretaciones, como lo sería interpretar probabilidades, pueden relacionarse con resultados subóptimos.

Capítulo 8

Resultados Reconocimiento de Expresiones

Ahora sí, usemos las técnicas desarrolladas para dar una solución concreta al problema de reconocer expresiones faciales sobre el dataset FER2013. Notemos que el experimento de elegir la etiqueta al azar tendría una exactitud de alrededor de $\frac{1}{7} \approx 0{,}14$.

8.1. Procesamiento de materiales

Hasta ahora en los laboratorios, estuvimos trabajando con datos de juguete que no necesitan de nuestra manipulación para directamente poderlos usar como input. En este caso si bien nos traemos de vuelta una base de datos ya procesada necesitaran algunas adaptaciones extras.

Para procesar los datos seguimos los siguientes pasos:

1. Interpretamos el vector de números de entrada como una matriz de píxeles de dimensión 48×48.

2. Usando la librería *Image* salvamos cada imagen en el directorio que contendrá las carpetas train/valid/test y las sub-carpetas según la emoción que le corresponda Enojo, Disgusto, Miedo, Felicidad, Tristeza, Sorpresa o Neutral.

3. Usaremos la función *image_dataset_from_directory* de pre-procesamiento propio de Keras para una mayor facilidad en el manejo de los datos.

El dataset FER2013 como dijimos proviene de la competencia en Kaggle, *Challenges in Representation Learning: Facial Expression Recognition*

Challenge, por ello tendremos una línea de referencia a la hora de caracterizar la potencia predictiva de los modelos: los **puestos según los puntajes obtenidos** por los participantes.

Para que las métricas sean comparables utilizaremos del dataset, los datos denotados en el Capítulo 3 como *training* para el entrenamiento, los *PublicTest* para la validación y los *PrivateTest* para el testeo. Las clases no están balanceadas pero se mantienen los porcentajes de cada una en cada set para ser consistentes.

8.2. Métodos y métricas

Los datos cuentan con imágenes centradas que facilitan la tarea pero, debido a la naturaleza intrínseca del problema (las distintas fisonomías; posibles formas de expresiones; posiciones de las caras, posibles ruidos dentro de las imágenes) es intuitivo pensar que estaremos en búsqueda de, nuevamente, un algoritmo que sea robusto a pequeñas traslaciones.

Recorramos los modelos vistos desde el principio de esta tesis y comparemos 3 arquitecturas:

1. **Regresión logística multinomial**: computamos la discriminación logística multiclase presentada en el Capítulo 4. Es evidente que este modelo no es adecuado para la tarea pero a fines de ir creciendo en complejidad de estructuras lo observaremos.

2 **Redes Fully Connected**: computamos una red neuronal clásica FC, presentada en el Capítulo 5, conformada por 4 capas con función de activación ReLU de 288, 128, 64 y 64 neuronas respectivamente. A esto le agregamos al final una capa de Batch Normalization y una de dropout con $\phi = 0{,}25$ como regularización.

3 **Redes Convolucionales**: computamos una red neuronal convolucional, presentada en el Capítulo 6, conformada por 4 capas convolucionales con función de activación ReLU usando 64, 64, 256 y 256 filtros respectivamente, todos de tamaño 3×3, stride 1 y same padding. Seguido de cada una de ellas agregamos una capa de Max Pooling de tamaño 3 con stride 2 y valid padding. A esto le agregamos nuevamente capas de regularización de Batch Normalization y Dropout con $\phi = 0{,}25$.

Las arquitecturas y desglose de parámetros correspondientes a cada una pueden encontrarse en el *Apéndice*. El código completo puede consultarse en

el siguiente repositorio de *Github*: *https://github.com/MaiteAngel/Clasificacion-Multinomial-con-Redes-Neuronales.git*

Para entrenar el modelo logístico mantuvimos nuevamente los parámetros sugeridos para su implementación en Python a través de la librería *sklearn*. Para compilar las arquitecturas de redes utilizamos una vez más el optimizador de Adam descripto en el Laboratorio de CNN y la función de pérdida será la sparse-cross-entropy multiclase. En general volveremos a manejarnos con la métrica de exactitud.

Los puntajes alcanzados en el set de testeo, en relación a los puestos obtenidos dentro de la competencia, se ven reflejados en la siguiente tabla:

Técnica	Exactitud	Posición competencia
Logístico	0.33	48-49
Redes clásicas	0.39	41-42
Redes convolucionales	0.66	4-5

Figura 8.1: Tabla de resultados

Las redes convolucionales parecen estar aprendiendo mucho mejor el problema. Para acercarnos a ganar los 500 dólares, que ofrecía la competencia al ganador, tenemos que ir escalando hacia arquitecturas cada vez más complejas y robustas como habíamos anticipado.

Para ofrecerle al lector una posible opción de cómo podría haber estado en el primer puesto, daremos un acercamiento que notamos como **Redes preentrenadas**. Se basa en la idea de importar arquitecturas ya entrenadas y adaptarlas al problema en cuestión. Es decir, nos traeremos la arquitectura y el resultado de los parámetros aprendidos de algún modelo que ya haya sido entrenado en una tarea similar. Los tomaremos como una estructura fija por la cual pasarán nuestros inputs.

En este caso haremos uso de *VGGFace2*. En el paper *VGGFace2: A dataset for recognising faces across pose and age* de los autores *Qiong Cao, Li Shen, Weidi Xie, Omkar M. Parkhi y Andrew Zisserman* se describe a VGGFace2 como un dataset enorme que contiene 3.31 millones de imágenes de 9131 sujetos distintos descargadas a través de Google Image Search. Se menciona que las redes convolucionales muy profundas, en particular la arquitectura conocida como Resnet-50 (una reconocida red residual constituida por capas convolucionales, de pooling y fully connected) entrenadas sobre este dataset se puede decir que son uno de los estados del arte en to-

das las tareas que involucren al fenómeno de *Face Recognition*: la tarea de identificar, verificar rostros.

En estas líneas, al estar buscando detectar expresiones en rostros parece una buena opción de un problema similar para adquirir sus enseñanzas.

Para importar el módulo utilizaremos un proyecto third-party open-source que es la conversión al ambiente de Keras del modelo *resnet50-VGGFace2* . Hacemos la llamada con la siguiente línea:

```
from keras_vggface.vggface import VGGFace

img_height, img_width = 224,224
module=VGGFace(model = 'resnet50',include_top = False,
weights = 'vggface',input_shape = (img_height, img_width, 3))
```

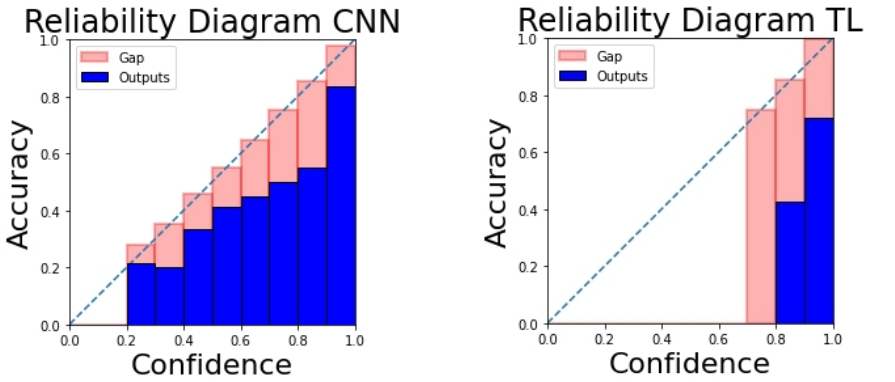
Procesamos los datos en función de esta elección (utilizamos el pre-procesador de Keras para hacerle el *reshape* correspondiente a la imagen), nos traemos los pesos ya calculados para esa arquitectura y simplemente le agregaremos, luego del modulo, algunas capas que lo ayuden a adaptarse al problema particular que intentamos resolver. En estas últimas sí aprenderemos los parámetros en la etapa de entrenamiento. En este caso sumamos una capa fully connected de 1024 neuronas y una extra de average pooling. Este proceso suele conocerse con fine-tunning y el fenómeno de usar este tipo de redes con pesos fijos no entrenables, Transfer Learning. Con esto podemos agregar un nuevo resultado a la tabla 8.1:

Técnica	Exactitud	Posición competencia
Logístico	0.33	48-49
Redes clásicas	0.39	41-42
Redes convolucionales	0.66	4-5
Redes preentrenadas	0.72	1

Figura 8.2: Tabla de resultados final

En sentido de la discusión del último capítulo, podemos realizar los *reliability diagrams* para las redes de mayor accuracy: la CNN y la red preentrenada(TL). Recordemos que lo que queríamos analizar es la *calibración* de los modelos. Para esto debemos contrastar en distintos intervalos del $[0, 1]$, los notados B_m, el promedio de la exactitud de las muestras que tengan una confianza de predicción dentro de B_m ($acc(Bm)$), contra el promedio de

estas confianzas ($conf(Bm)$).Si todas las barras coinciden en la recta identidad conseguiríamos una calibración perfecta (objetivo inalcanzable en la práctica).



((a)) *RD para la red CNN* **((b))** *RD para la red preentrenada*

Figura 8.3: Reliability diagrams (RD)

En esta representación, para cada B_m el azul describe el menor valor alcanzado, es decir el $min(acc(Bm), conf(Bm))$. El color rosa representa la distancia que hay entre la exactitud y la confianza, es decir $|acc(Bm) - conf(Bm)|$, es el "gap" entre ambas.

Cabe aclarar que una limitación de este gráfico es que no se muestra la cantidad de observaciones que caen dentro de cada intervalo. Luego, por ejemplo en la figura *8.3 ((b))* el intervalo $(0{,}7, 0{,}8]$ tiene una sola observación con confianza 0.75 y exactitud 0.

Podemos observar que la red convolucional parece tener un rango de confianzas de predicción amplio pero las distancias a la exactitud son muy marcadas. Esto nos dice que a pesar de predecir muy bien, no es tan buena describiendo su capacidad de acierto.

Esto se acentúa todavía más en la red preentrenada, es muy confiada en sus predicciones no alcanzando menores a 0.7.

Por último tomamos la red CNN y le aplicamos Temperature Scaling para mostrar un modelo calibrado que mantiene su métrica. Recordemos que solo debemos optimizar por verosimilitud un escalar que suavizará las confianzas repartiéndolas mejor a lo largo del $[0, 1]$. Logramos el siguiente diagrama:

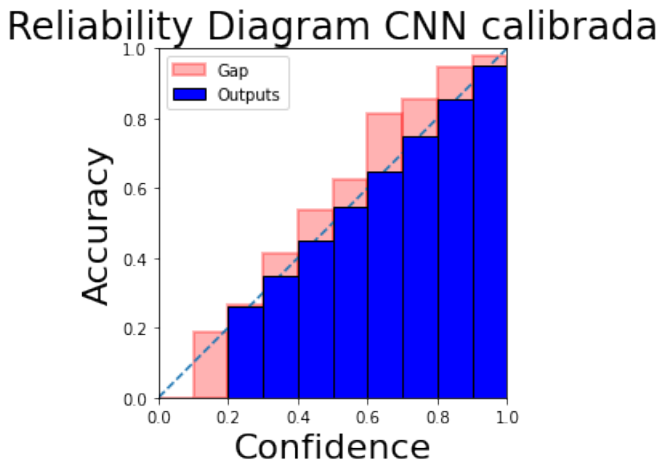


Figura 8.4: "Reliability diagram para la red CNN con Temperature Scaling'

Ahora las diferencias son mucho menores. Tan solo tuvimos que reinterpretar la salida y entrenar un parámetro, en contraposición a descartar todo o parte de lo ya aprendido.

8.3. Exploración de resultados

Ahora observemos los desempeños de cada uno de los modelos más en detalle en los siguientes mapas de matrices de confusión.

Las matrices de confusión M son una manera establecida de visualizar los errores de cada una de las clases. En las imágenes que siguen las definimos de forma que:

- $M_{i,j}$:= La frecuencia con la que el modelo, sobre el set de imágenes de testeo, predijo la etiqueta j cuando la verdadera etiqueta era i.

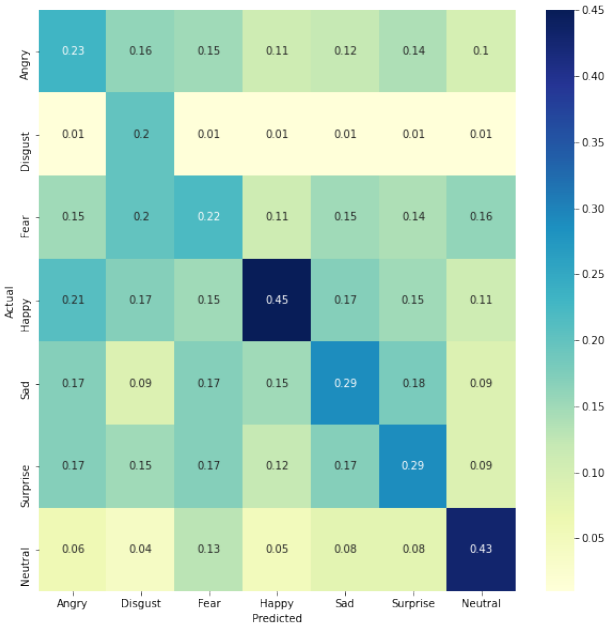


Figura 8.5: "Matriz de confusión Regresión Logística"

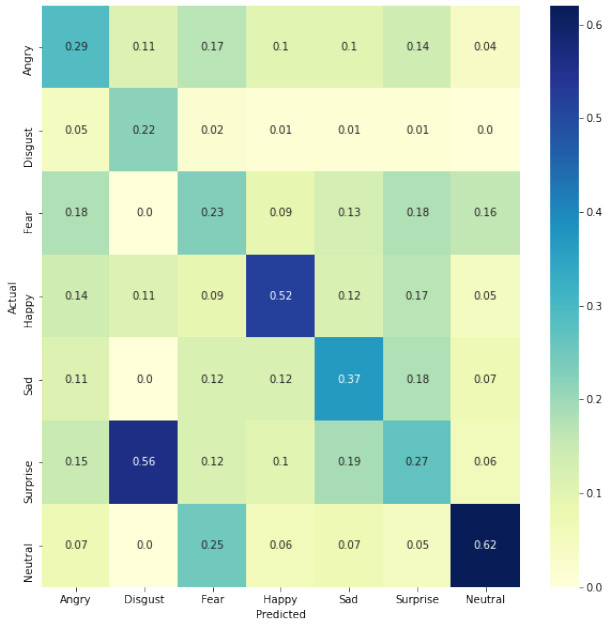


Figura 8.6: "Matriz de confusión Redes FC"

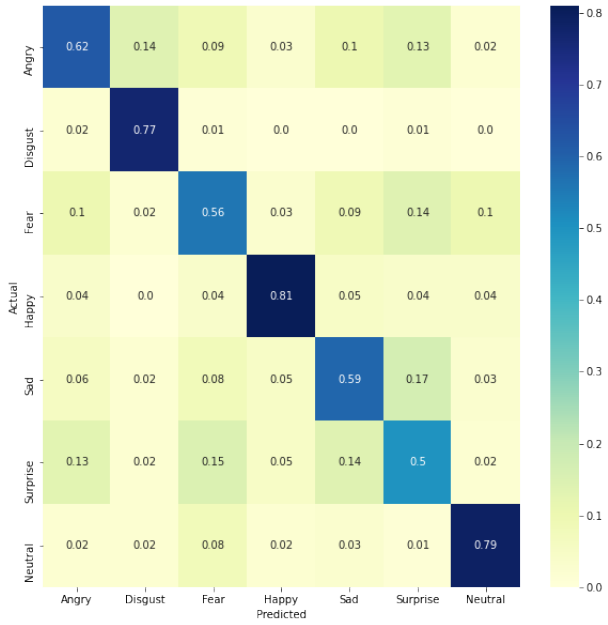


Figura 8.7: "Matriz de confusión Redes CNN"

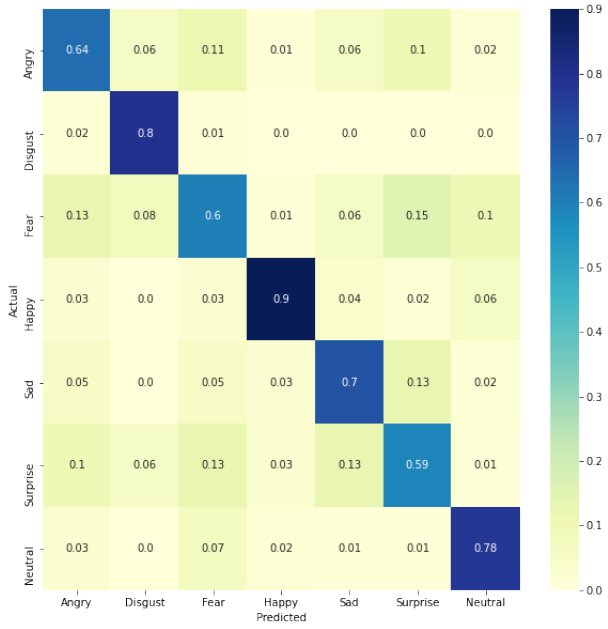


Figura 8.8: "Matriz de confusión Redes preentrenadas'

Algo que puede observarse es por ejemplo que las redes que alcanzaron mejores resultados no se equivocan nunca la emoción Disgusto (3) con la

Felicidad (1) y la Felicidad con Disgusto solo 3 veces en el caso de las CNN y 1 vez en el caso de las redes preentrenadas. También la Felicidad parece ser una de las emociones más fáciles de reconocer. Veamos un sampleo de las imágenes que pertenecen a estas emociones:

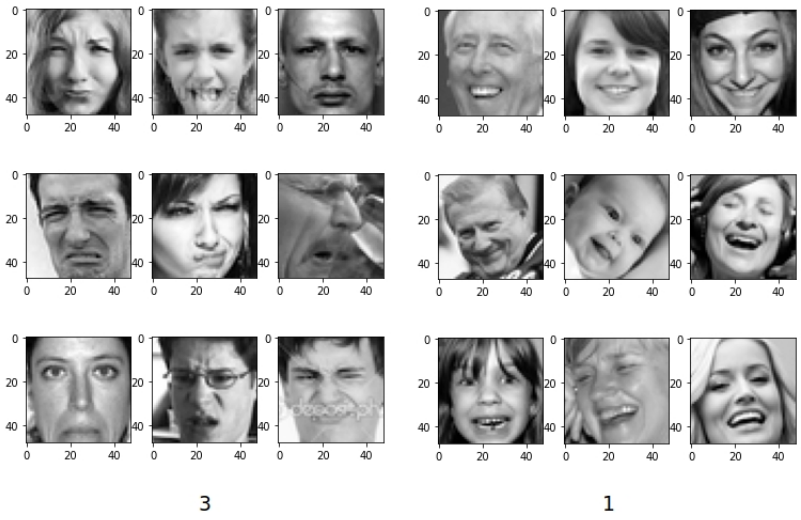
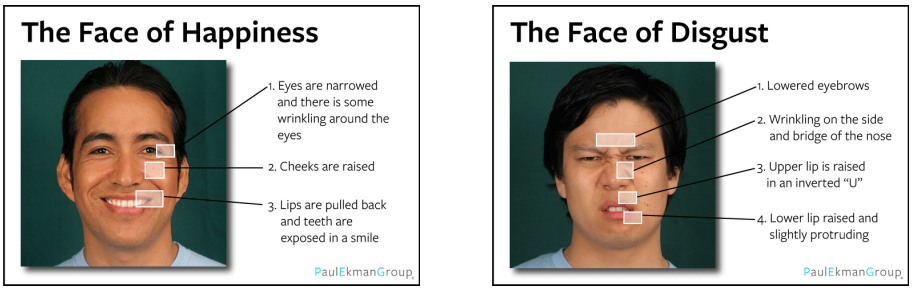


Figura 8.9: "Expresiones de Disgusto(3) vs Felicidad(1)"

Paul Ekman en algunos de sus estudios como *Universals and Cultural Differences in Facial Expressions of Emotions (1972)* y *Universal Facial Expressions of Emotions (1970)* analiza estas mismas expresiones, a las que llama emociones universales, dando ciertas referencias de cuáles son los puntos faciales que las caracterizan, por ejemplo para las expresiones de disgusto y felicidad tenemos las siguientes observaciones:



((a)) *La cara de la felicidad* **((b))** *La cara del disgusto*

Figura 8.10: Paul Ekman Group: Signos de las emociones.

En sus trabajos se nota que el signo característico de la expresión de

felicidad es la sonrisa. La expresión de disgusto, por su parte, es fácilmente reconocible sobre todo por arrugar la nariz. Son ambas visualmente muy distintas, mientras que la expresión de disgusto tiende a concentrar los rasgos alrededor de la nariz, la felicidad parecería inclinarse por ampliarlos.

Estos patrones en las caras son lo que aspiramos a que los modelos aprendan e interpreten correctamente. Si viéramos estas imágenes como lo ve la red convolucional sería algo como lo siguiente,

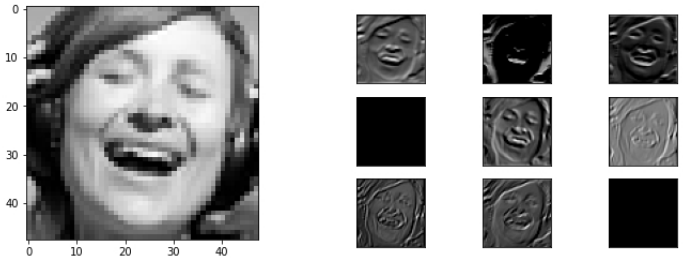


Figura 8.11: "Imagen original vs feature maps que corresponde a Felicidad"

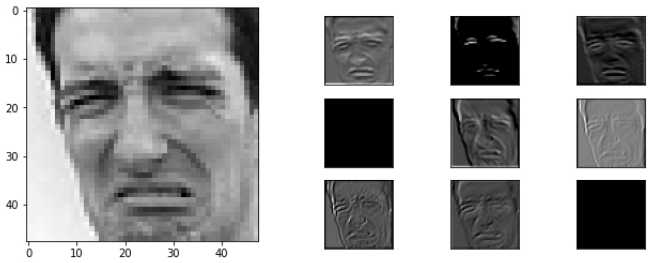
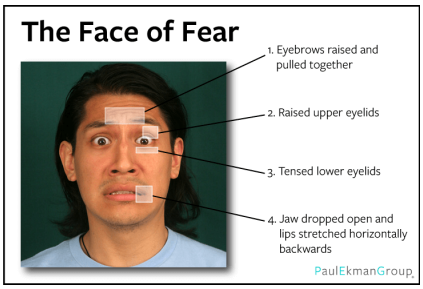


Figura 8.12: "Imagen original vs feature maps que corresponde a Disgusto"

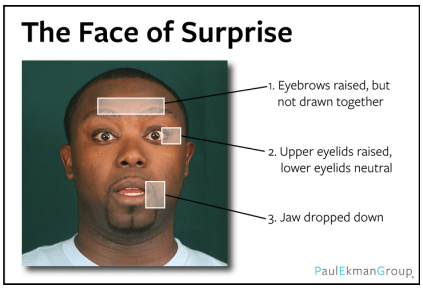
Estas salidas son algunos de los feature maps que se obtienen luego de haber convolucionado por los pesos encontrados.

Vemos como va encontrando las líneas más relevantes alrededor de la boca, la nariz y los ojos consistentes con lo que nos marcaba el estudio antes mencionado.

Por último también podemos observar que las expresiones que más parecen confundirse son la sorpresa y el miedo, por lo que esperaríamos que los rasgos que las identifican sean parecidos. En efecto, vayamos directo a las imágenes que muestran sus signos faciales característicos:



((a)) *La cara del miedo*



((b)) *La cara de la sorpresa*

Figura 8.13: Paul Ekman Group: Signos de las emociones.

Siguiendo este mismo estudio, se destacan estas dos expresiones como uno de los pares que más suelen confundirse debido a que se manifiestan en los mismos puntos claves: cejas, ojos y boca.

Es interesante observar en estas líneas, que las redes convolucional y preentrenada, que se destacaron en términos de exactitud, se equivocan mayoritariamente en clases que son difíciles de distinguir naturalmente. Si la tarea fuese realizada por el ojo humano, en condiciones razonables, este estudio nos indica que serían factibles de confundirse.

Capítulo 9

Conclusiones

En esta tesis, centrándonos en el problema de clasificación, partimos de los modelos estadísticos más clásicos, como lo es la regresión logística, para luego generalizar hacia propuestas más abstractas. Llegamos así a la definición formal de las redes Fully Connected para luego hacer el salto y extenderlas hacia arquitecturas aún más complejas como lo son las redes Convolucionales.

En *primer lugar* se la puede considerar como aporte a la recopilación y formalización de una descripción teórica de las operaciones matemáticas involucradas en el mundo de la clasificación binaria y, sobre todo, multinomial para las técnicas de redes neuronales desde un punto de vista probabilístico. Hemos podido identificar el salto del mundo estadístico hacia el del aprendizaje automático como la pérdida de identificabilidad de parámetros para tomar un camino donde se permite la sobreparametrización. Hemos detectado, como un punto clave en la confección moderna de los modelos, que el objetivo se centra en la predicción; por esto la calibración de los modelos pasa a un segundo plano. Luego, cuando se discuten posibles interpretaciones de las salidas, hay que tener en cuenta que se pueden obtener resultados subóptimos a pesar de tener muy buenas métricas en términos de discriminación.

En *segundo lugar* hemos logrado crear, en las secciones de laboratorio, ambientes controlados donde pudimos ofrecer escenarios donde se puede apreciar la potencia de saltar hacia estructuras más complejas:

- En primera instancia, el salto de la discriminación logística a redes FC con una capa oculta levantando el supuesto de linealidad de los datos. Mostramos en un ejemplo práctico como la necesidad de un proceso

de Feature engineering, donde se deben conocer las particularidades de los datos para manipularlos correctamente, se levanta con las redes neuronales FC donde este proceso se vuelve implícito a la hora del entrenamiento.

- En segunda instancia creciendo hacia redes convolucionales, sobre el dataset MINST de reconocimiento de dígitos, entrenamos dos arquitecturas, una red FC y una CNN donde ambas se desempeñaron razonablemente bien en términos de predicción. Luego corrompimos el conjunto de datos de testeo, permitiendo pequeñas traslaciones de los objetos que nos dejaron mostrar un escenario donde las FC perdían todo su poder, empeorando mucho en términos de exactitud (0.3), en tanto las CNN tuvieron un desempeño mucho más robusto, mostrándose más invariantes a estos cambios (0.7).

En *tercer lugar*, en el problema de reconocimiento de expresiones faciales sobre el dataset FER2013, hemos podido adaptar una implementación de las técnicas de mayor interés. Haciendo uso de la reconocida técnica de Transfer Learning hemos podido ofrecer al lector un modelo que logra una métrica que alcanzaría el primer puesto de la competencia.

En *último lugar* se aportó código replicable en Python, disponibilizando el código completo en el repositorio de Github:
https://github.com/MaiteAngel/Clasificacion-Multinomial-con-Redes-Neuronales
Se aportaron soluciones al problema particular del Reconocimiento de expresiones faciales y se dieron ejemplos claros y controlados donde se observan los beneficios concernientes a la generalización que se adquiere con el uso de estructuras más complejas.

Finalmente como trabajo a futuro queda poder explorar, evaluar y contrastar contra las redes neuronales vistas en este estudio algunas técnicas estadísticas más avanzadas como lo es *support vector machine (SVM)*. También se abre paso a describir otros tipos de redes como las recurrentes, que fueron mencionadas brevemente en el estudio. Además de explorar otros marcos de problemas dentro del fenómeno de aprendizaje automático, más específicamente en el campo de *Computer Vision*. Creemos que también dejamos la puerta abierta a una gran e importante discusión como lo es el equilibrio de conseguir modelos que discriminen bien y también estén bien calibrados.

Bibliografía

[1] Hastie.T, Witten.D, Tibshirani.R, James.G (2013): An introduction to Statical Learning with Applications in R. Springer Texts in Statistics

[2] Agresti A. (2015): Foundations of linear and generalized linear models. John Wiley & Sons.

[3] B. D.Ripley (1994): Neural Networks and Related Methods for Classification. Source: Journal of the Royal Statistical Society. Series B (Methodological), Vol. 56, No. 3 , pp. 409-456 Published by: Wiley for the Royal Statistical Society Stable.

[4] Cybenko, G. (1989): Approximation by superpositions of a sigmoidal function. Math. Control Signal Systems 2, 303–314 .

[5] Alizadeh, Shima Fazel, Azar. (2017): Convolutional Neural Networks for Facial Expression Recognition.

[6] Kurt Hornik, Maxwell Stinchcombe, Halbert White (1989): Multilayer feedforward networks are universal approximators. Volume 2, Issue 5, Pages 359-366.

[7] Jerome H. Friedman and Werner Stuetzle (1981): Projection Pursuit Regression. Source: Journal of the American Statistical Association, Vol. 76, No. 376 , pp. 817- 823 Published by: American Statistical Association

[8] Bradley Efron and Trevor Hastie (2016): Computer Age Statistical Inference Algorithms, Evidence, and Data Science.

[9] Michael A. Nielsen (2015): Neural Networks and Deep Learning. Determination Press.

[10] Jason Brownlee. (2021): Machine Learning Mastery Pty. Ltd. All Rights Reserved.

[11] Koushik, Jayanth (2016): Understanding Convolutional Neural Networks.

[12] Ian Goodfellow, Yoshua Bengio and Aaron Courville (2016): Deep Learning. MIT Press.

[13] Zhang, Z. (2016): 'Derivation of Backpropagation in Convolutional Neural Network (CNN)'

[14] Q. Cao, L. Shen, W. Xie, O. M. Parkhi and A. Zisserman (2018): VGGFace2: A Dataset for Recognising Faces across Pose and Age, 2018 13th IEEE International Conference on Automatic Face Gesture Recognition (FG 2018), pp. 67-74, doi: 10.1109/FG.2018.00020. 423-443.

[15] Goodfellow I.J. et al. (2013): Challenges in Representation Learning: A Report on Three Machine Learning Contests. In: Lee M., Hirose A., Hou ZG., Kil R.M. (eds) Neural Information Processing. ICONIP 2013. Lecture Notes in Computer Science, vol 8228. Springer, Berlin, Heidelberg.

[16] Guo, Chuan Pleiss, Geoff Sun, Yu Weinberger, Kilian (2017): On Calibration of Modern Neural Networks.

[17] Luciana Ferrer (2019): Machine Learning Fundamentals with a Focus on Evaluation Practices. Khipu.

[18] Paul Ekman paulekman.com/universal-emotions Copyright © 2021 Paul Ekman Group LLC.

[19] Andrew Ng. et al. (n.d.): Convolutional Neural Networks. Coursera.

Apéndice

Regresión logística

```
model = LogisticRegression(multi_class='multinomial',fit_intercept=True,
        max_iter=10000)
model.fit(x_train, y_train)

---------------------------------------------------------------------
Pipeline(memory=None,
         steps=[('standardscaler',
                 StandardScaler(copy=True, with_mean=True, with_std=True)),
                ('logisticregression',
                 LogisticRegression(C=1.0, class_weight=None, dual=False,
                                    fit_intercept=True, intercept_scaling=1,
                                    l1_ratio=None, max_iter=10000,
                                    multi_class='multinomial', n_jobs=None,
                                    penalty='l2', random_state=None,
                                    solver='lbfgs', tol=0.0001, verbose=0,
                                    warm_start=False))],
         verbose=False)
```

Redes FC

```
model_fc = keras.Sequential([
    keras.layers.InputLayer(input_shape=(48,48, 1)),
    keras.layers.Lambda(lambda x: x/255.0),
    keras.layers.Flatten(),
    keras.layers.Dense(288, activation='relu',),
    keras.layers.Dense(128, activation='relu'),
    keras.layers.Dense(64, activation='relu'),
    keras.layers.Dense(64, activation='relu'),
    keras.layers.BatchNormalization(),
    keras.layers.Dropout(0.25),
```

```
    keras.layers.Dense(7, activation='softmax')
])
model_fc.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lambda_2 (Lambda)            (None, 48, 48, 1)         0
_________________________________________________________________
flatten_2 (Flatten)          (None, 2304)              0
_________________________________________________________________
dense_6 (Dense)              (None, 288)               663840
_________________________________________________________________
dense_7 (Dense)              (None, 128)               36992
_________________________________________________________________
dense_8 (Dense)              (None, 64)                8256
_________________________________________________________________
dense_9 (Dense)              (None, 64)                4160
_________________________________________________________________
batch_normalization_3 (Batch (None, 64)                256
_________________________________________________________________
dropout_3 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_10 (Dense)             (None, 7)                 455
=================================================================
Total params: 713,959
Trainable params: 713,831
Non-trainable params: 128
```

Redes CNN

```
model = keras.Sequential([
    keras.layers.InputLayer(input_shape=(48,48, 1)),
    keras.layers.Lambda(lambda x: x/255.0),

    keras.layers.Conv2D(64, (3, 3), activation='relu', padding = 'same'),
    keras.layers.MaxPooling2D(pool_size=(3,3), strides=(2, 2)),
    keras.layers.BatchNormalization(),
    keras.layers.Dropout(0.25),
```

```
    keras.layers.Conv2D(64, (3, 3), activation='relu', padding = 'same'),
    keras.layers.MaxPooling2D(pool_size=(3,3), strides=(2, 2)),
    keras.layers.BatchNormalization(),
    keras.layers.Dropout(0.25),

    keras.layers.Conv2D(256, (3, 3), activation='relu', padding = 'same'),
    keras.layers.MaxPooling2D(pool_size=(3,3), strides=(2, 2)),
    keras.layers.BatchNormalization(),
    keras.layers.Dropout(0.25),

    keras.layers.Conv2D(256, (3, 3), activation='relu', padding = 'same'),
    keras.layers.MaxPooling2D(pool_size=(3,3), strides=(2, 2)),
    keras.layers.BatchNormalization(),
    keras.layers.Dropout(0.25),

    keras.layers.Flatten(),
    keras.layers.Dense(7, activation='softmax')
])
model.summary()

Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lambda_1 (Lambda)            (None, 48, 48, 1)         0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 48, 48, 64)        640
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 23, 23, 64)        0
_________________________________________________________________
batch_normalization_16 (Batc (None, 23, 23, 64)        256
_________________________________________________________________
dropout_16 (Dropout)         (None, 23, 23, 64)        0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 23, 23, 64)        36928
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 11, 11, 64)        0
_________________________________________________________________
batch_normalization_17 (Batc (None, 11, 11, 64)        256
_________________________________________________________________
dropout_17 (Dropout)         (None, 11, 11, 64)        0
```

```
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 11, 11, 256)       147712
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 5, 5, 256)         0
_________________________________________________________________
batch_normalization_18 (Batc (None, 5, 5, 256)         1024
_________________________________________________________________
dropout_18 (Dropout)         (None, 5, 5, 256)         0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 5, 5, 256)         590080
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 2, 2, 256)         0
_________________________________________________________________
batch_normalization_19 (Batc (None, 2, 2, 256)         1024
_________________________________________________________________
dropout_19 (Dropout)         (None, 2, 2, 256)         0
_________________________________________________________________
flatten_11 (Flatten)         (None, 1024)              0
_________________________________________________________________
dense_51 (Dense)             (None, 7)                 7175
=================================================================
Total params: 785,095
Trainable params: 783,815
Non-trainable params: 1,280
```

Redes Preentrenadas

```
img_height, img_width = 224,224
module=VGGFace(model = 'resnet50',include_top = False,
weights = 'vggface',input_shape = (img_height, img_width, 3))

model = keras.Sequential([
    keras.layers.InputLayer(input_shape=(img_height, img_width, 3)),
    keras.layers.Lambda(lambda x: (x-128.8006)),
    module,
    keras.layers.AveragePooling2D(pool_size=(2,2), strides=(2, 2),padding="sa
    keras.layers.Flatten(),
    keras.layers.Dense(1024, activation="relu"),
    keras.layers.Dense(7, "sigmoid"),
])
```

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lambda (Lambda)              (None, 224, 224, 3)       0
_________________________________________________________________
vggface_resnet50 (Functional (None, 1, 1, 2048)        23561152
_________________________________________________________________
average_pooling2d (AveragePo (None, 1, 1, 2048)        0
_________________________________________________________________
flatten (Flatten)            (None, 2048)              0
_________________________________________________________________
dense (Dense)                (None, 1024)              2098176
_________________________________________________________________
dense_1 (Dense)              (None, 7)                 7175
=================================================================
Total params: 25,666,503
Trainable params: 25,613,383
Non-trainable params: 53,120
```