Instituto Tecnológico Autónomo de México

Programación con Redes Neuronales

Tesis

que para obtener el título de

Licenciado en Matemáticas Aplicadas

presenta

Diego Alberto Aguado Hernández

Asesor: Dr. Rodrigo Mendoza Smith

México, Ciudad de México. 2019

Autorización

Con fundamento en el artículo 21 y 27 de la Ley Federal del Derecho de Autor y como titular de los derechos moral y patrimonial de la obra titulada "Programación con Redes Neuronales", otorgo de manera gratuita y permanente al Instituto Tecnológico Autónomo de México y a la Biblioteca Raúl Bailléres Jr. autorización para que fijen la obra en cualquier medio, incluido el electrónico y la divulguen entre sus usuarios, profesores, estudiantes o terceras personas, sin que pueda percibir por la divulgación una contraprestación.

Diego Alberto Aguado Hernández

Fecha

Firma

Para mis papás, Jorge y Lucía, mi definición personal de perseverancia.

Agradecimientos

Somos el, extraño, colectivo de las interacciones que tenemos con el mundo y, sobre todo, otros seres humanos. Este trabajo representa la culminación de una gran etapa de mi vida. Una etapa llena de interacciones tanto emocionantes como determinantes y a veces extrañas. Gracias a todas las personas que en este viaje influenciaron mi tren de pensamiento matemático y de cosmovisión, de forma explícita o implícita. Y aunque las personas a las que quisiera agradecer son más de las que caben en esta página, hay un conjunto de ellas con cardinalidad pequeña que merecen más que un gracias, una ovación. Mis padres y hermanos, que nunca imaginaron que estudiaría algo, en palabras de mi padre, tan árido. Sin embargo, siempre me apoyaron y creyeron que haría mi mejor esfuerzo. Mi asesor Rodrigo Mendoza Smith, que en momentos de frustración me hizo creer en mi mismo de inusuales maneras, siempre encontrando la forma de motivarme. A mis amigos, los de toda la vida y los que conocí durante estos años que ahora son para la prosperidad, por ser indispensables en este proceso también.

Índice general

Capítulo 1

Inducción de programas

El objetivo de ésta tesis es presentar una revisión literaria y un caso práctico de modelos de aprendizaje de máquina para *programación inducida*, específicamente para generación de código a partir de imágenes. La *programación inducida* se refiere al conjunto de técnicas en teoría de programación e inteligencia artificial para la generación de programas declarativos a partir de especificaciones incompletas o abstractas. En los últimos años, esta área ha crecido de manera considerable gracias a los avances en *aprendizaje profundo* y a la disponibilidad de repositorios masivos de código, que permiten la recolección sistematizada de código para ser ingestados por modelos de aprendizaje de máquina y, más específicamente, redes neuronales. Los datos usados para inducción de programas con aprendizaje de máquina son similares a los usados en procesamiento de lenguaje y considerablemente más estructurados que los datos usados en visión computacional o robótica. Es decir, los lenguajes de programación pueden ser analizados

con herramientas provenientes del procesamiento de lenguaje natural, pero exhiben otros retos.

Por ejemplo, los lenguajes de programación tienen sintaxis y gramáticas estáticas y definidas a priori por las reglas de compilación del lenguaje. Este no es el caso del lenguaje humano o natural. Este dispone de un gran número de figuras literarias y gramaticales que dotan de flexibilidad a la lengua, pero al costo de estructura. Aun más, en este una palabra puede tomar un significado distinto dado el contexto, esto no es característico de los lenguajes de programación. Así el estudio del procesamiento de lenguaje natural y de lenguajes de programación es en gran parte un ejercicio en el cual se deben saber combinar conceptos de gramática, sintaxis, y semántica.

El caso práctico en el que se centra este trabajo es el propuesto en [Beltramelli, 2017], el mapeo de interfaces gráficas al código que da lugar a estas. Esto es, mapear datos con estructura de rejilla a datos secuenciales. Las naturalezas propias de cada conjunto son particularmente relevantes ya que, como se precisa más adelante en el trabajo, deben ser procesadas por algoritmos distintos. En el trabajo replicado, se estudia el modelo *pix2code* para resolver este problema. Este modelo es una red neuronal que toma en cuenta las naturalezas de los dos conjuntos de datos, imágenes y código. Esta tesis hace una deconstrucción del modelo y desarrolla de manera teórica los componentes necesarios para reconstruirlo. Así mismo, se proponen dos modelos alternativos para la resolución del mismo problema, mismos que presentan mejoras en las métricas analizadas.

En este capítulo se describe el trasfondo histórico que llevó al desarrollo de las ciencias computacionales y de la Inteligencia Artificial (IA) para

describir el contexto en el que se gesta la programación inducida. Seguido de esto, se presentan los inicios de esta linea de investigación así como trabajos iniciales en el tema. Después, se detalla el desarrollo de los acontecimientos que dieron lugar a los avances en aprendizaje de máquina durante los años 2000 y principios del 2010. Finalmente, se presentan algunos trabajos y nuevos enfoques a programación inducida haciendo uso de aprendizaje de máquina e IA.

1.1. Orígenes de la Inteligencia Artificial y Programación Inducida

La capacidad de analizar y resolver distintos problemas de alta complejidad ha sido el factor determinante para el crecimiento de la humanidad. Un comportamiento característico de los seres humanos es su capacidad de automatizar procesos. A lo largo de la historia, se ha buscado eficientar la realización de tareas repetitivas y enagenantes a través de su automatización. Sin embargo, en esta búsqueda no era extraño encontrarse con una barrera natural. Esta siempre fue la necesidad de consultar el criterio de una persona, su inteligencia.

Por otro lado, con el invento de la computadora, el ser humano expandió su capacidad no solo de almacenamiento y procesamiento de datos sino también su capacidad creativa. Con la llegada de la computadora, se abrió la puerta a un universo abstracto en el cuál el hombre puede estructurar su pensamiento lógico, crear posibilidades de visualizaciones infinitas con esfuerzos marginales o establecer comunicación entre millones de individuos

a niveles nunca antes imaginados. Desde sus inicios, los sistemas computacionales fueron diseñados con las matemáticas como fundamento de estos. Así, las ciencias computacionales se convirtieron en objeto de estudio casi inmediato de las matemáticas.

Con la interacción de la constante automatización de tareas y la era de la computación, la comunidad científica empezó a preguntarse sobre la viabilidad de automatizar tareas dentro del campo de la computación, algoritmos para la creación de programas. Sin embargo, era claro que para crear un sistema automatizado de programación sería necesario dotarlo de cierta inteligencia. Para tan solo pensar en capacidades o inteligencia de sistemas, hay que retroceder en el tiempo a 1950. En este año, Alan Turing conceptualizó y formalizó la idea de inteligencia en computadoras.

1.1.1. La conceptualización de inteligencia en computadoras

Alan Mathison Turing fue un lógico-matemático inglés. Aunque es conocido principalmente por su participación en la creación de la computadora *Bombe* durante la segunda guerra mundial, fue uno de los pioneros más importantes en el desarrollo de la computación. Es considerado por algunos como el padre de esta. El trabajo de Turing era práctico y teórico, definió las bases de la computación y, como menciona [Muggleton, 2014], predijo alcances de Inteligencia Artificial, rama de la ciencia que se desarrollaría hasta años después.

Turing, en su trabajo [Turing, 1950] provoca el pensamiento sobre la posibilidad de crear máquinas que pudieran *pensar*. Para discernir si una máquina tiene verdadera inteligencia Turing diseñó una prueba que lleva su

nombre. La *prueba de Turing* puede ser formulada de la siguiente manera. Supongamos que un interrogador humano hace una serie de preguntas a una máquina. Si esta puede sostener una conversación con el interrogador sin que el primero pueda discernir entre máquina o humano, entonces esta ha pasado la prueba. En una época en la que las computadoras apenas habían sido inventadas y cuyo propósito era efectuar cálculos y procesamiento, Alan Turing plantea la pregunta "¿Las computadoras pueden pensar?". Esta pregunta define el contexto en el cual nace la Inteligencia Artificial.

El término Inteligencia Artificial fue acuñado en la Universidad de Dartmouth durante el evento "Proyecto de investigación de Verano sobre Inteligencia Artificial". Este, surgió del clima de desarrollo en matemáticas, ciencias computacionales, psicología, entre otras. Convocado en el verano de 1956, el proyecto fue iniciado por John McCarthy, Marvin Minsky, Nathaniel Rochester y Claude Shannon. Con distintas formaciones, el objetivo de los científicos era claro. Descrito en [McCarthy and et al, 1955], el objetivo tenía una clara influencia de Turing: "El estudio se realizará fundamentándose en la conjetura de que todos los aspectos de aprender, o cualquier otro distintivo de inteligencia, puede, en principio, ser tan precisamente descrito que una máquina pueda ser construida para simularlo.". Este objetivo reflejaba la forma en que Turing conceptualizó la forma de emular el pensamiento inteligente, humano en máquinas.

Segun [Norvig and Russel, 2003], el taller no dio a luz ningún resultado o avance nuevo. Sin embargo, logró crear una comunidad que dominaría la escena en los años venideros y la convención de usar el nombre Inteligencia Artificial. De acuerdo con [Norvig and Russel, 2003], históricamente el

objeto de estudio de la Inteligencia Artificial ha tomado varios enfoques:

1. Entidades que piensan como humanos
2. Entidades que actuan como humanos
3. Entidades que piensan racionalmente
4. Entidades que actuan racionalmente.

Hipotetizamos que una entidad cuyo propósito está relacionada a programación inducida necesitaría contar los aspectos uno, tres y cuatro de los listados anteriormente. Curiosamente, el primer trabajo sobre redes neuronales fue propuesto en [McCulloch and Pitts, 1943], una simulación de una red neuronal con circuitos eléctricos. En los años siguientes, las computadoras empezaron a desarrollarse cada vez con mayor capacidad de cómputo, lo cual dio nuevas herramientas para investigación siendo usadas por casi todas las ramas de la ciencia. En particular las matemáticas, ingenierías, entre otras se vieron beneficiadas de manera fundamental.

En 1957, en el trabajo [Frank, 1957], el neurobiologo Rosenblat desarrolló el perceptrón (Mark I Perceptron): el primer pedazo de hardware capaz de *aprender* a base de prueba y error. La tarea del *Mark I* era reconocimiento de imágenes. Rosenblat probó el teorema de convergencia del perceptron. A pesar de que el perceptron prometía grandes capacidades de modelaje, no pasó mucho tiempo antes de que se diera a conocer que no podía ser entrenado para discernir muchos tipos de patrones. Esto causó que la investigación sobre redes neuronales se pausara por muchos años.

Ejemplos como estos fueron apareciendo en la época dorada de la Inteligencia Artificial, hasta que a principios de los años 70 llegó una época conocida como el primer invierno de la IA. Específicamente, la investigación sobre las redes neuronales fue detenida virtualmente de forma total debido a las aseveraciones hechas por Marvin Minsk en su libro *Perceptrons*. Algunos de los problemas principales que causaron esta desaceleración fueron el poder de cómputo (no había suficiente poder de almacenamiento o de procesamiento) y la, enorme, necesidad de conocimiento previo o razonamiento(tareas relacionadas con visión computacional o procesamiento de lenguaje natural).

1.1.2. La programación como estudio matemático

En los años 1950 con el desarrollo de las computadoras, se transformó el concepto de software. Dejó de considerarse como configuraciones mecánicas y lentamente empezó a tomar un carácter informativo, datos. Los compiladores y ecosistemas de ejecución son también software, por lo que la frontera entre software y datos empezaba a desvanecerse. La idea de I.A. en programación comenzó a filtrarse en la mente colectiva de la comunidad científica. Así, la pregunta de si el software podía programarse a sí mismo empezó a resonar en el medio e intrigó a matemáticos, científicos y filośofos. Esta pregunta estaba relacionada con el *problema de la interrupción* que había sido probado años antes por Alan Turing en su trabajo [Turing, 1936]. En este, demostró que no existe un algoritmo para probar que, dada la descripción de un programa y un dato de entrada, se puede determinar si el programa termina o corre por siempre.

En esa época, el estudio matematizado del software y sus propiedades fue un tema de interés y de detallado estudio, véase [Kleene, 1952]. En particular, mencionamos [Waldinger and Lee, 1969] donde se propone un programa que recibe la especificación de un programa en un lenguaje de lógica simbólica y produce un programa en lenguaje LISP junto con su implementación. En [Manna and Waldinger, 1975] se exploran los elementos que los programas para síntezar un segundo programa deben de cumplir, en particular la manera en que varias partes del programa deben interactuar entre ellas y la forma en que el programa debe satisfacer diferentes metas simultáneamente. [Summers, 1977] y [Bierman, 1978], consideran los primeros métodos para construir programas a partir de ejemplos. Este acercamiento empieza a expresar la idea de *aprendizaje supervisado* en programación inducida. El concepto de aprendizaje supervisado se explora en el Capítulo 2.

1.1.3. Los grandes datos en software

Con el surgimiento del *big data*, el problema de síntesis de programas, programación inducida y metaprogramación tomó una nueva dirección. En efecto, en la seguna década del siglo XXI los repositorios de código masivos se volvieron populares y proporcionaron una fuente de datos de código que antes era escasa. Esto hizo que se estudiaran nuevas maneras de generar código usando aprendizaje de máquina. Por ejemplo, en [Solar-Lezama, 2009], se exploran métodos de síntesis de software donde un procedimiento automatizado sintetiza un programa a partir de un conjunto *programas parciales* suministrados por el usuario para describir una estrategia de im-

plementación deseada.

En [Allamanis and Sutton, 2013], se hace uso de miles de lineas de código público para crear un modelo probabilístico que, al tiempo de su publicación, es considerablemente mejor para la tarea de sugerencia de código. Así mismo, se proponen nuevas métricas de complejidad de código y exponen la posibilidad de identificar secciones de código reutilizables. Este trabajo permite vislumbrar la posibilidad de modelos que "entienden" sobre código eficiente.

En [Raychev et al., 2014], se presenta un modelo que llena espacios en código incompleto haciendo uso de modelos de lenguaje estadístico. Para esto, se hace una recolección masiva de código haciendo uso de repositorios públicos. Este fue un gran recurso durante esta época dado que el acceso a grandes bases de datos de código representaba un gran reto anteriormente. Así mismo, en [Raychev et al., 2015] se presenta un modelo probabilístico capaz de predecir nombres de variables y anotaciones sobre el tipo de variables. Esto refleja un correcto modelaje a nivel sintáctico y semántico, respectivamente.

En el trabajo [Bichsel et al., 2016], se vuelve a hacer uso de modelos lenguaje para limpieza y simplificación de código usado en aplicaciones Android. El problema de simplificación se plantea como la predición de un modelo gráfico probabilístico que fue entrenado bajo un conjunto de caracterizaciones y reglas a considerar.

En otro trabajo relacionado, [Menon et al., 2013] ataca el problema de creación de código. La manera de hacer esto es creando un sistema que "aprende" de observar pares entrada-salida buscando una serie de compo-

siciones de funciones para lograr el resultado correcto.

1.1.4. Aprendizaje Profundo, nuevos enfoques

La idealización de una verdadera IA está aun lejos de lo alcanzado en la práctica. El uso de funciones matemáticas, lógicas o selección de variables son ejemplos de bloques constructivos para escribir código. ¿Se puede crear un algoritmo que tenga la capacidad de resolver problemas escribiendo código en algún lenguaje computacional? Es ahí, donde se encontró una frontera en la cual las redes neuronales fueron modificadas para crear sistemas con capacidades más complejas. Actualmente existen grandes promesas de empujar la utilización de estos algoritmos para el aprendizaje de cómputo y programación.

Con el crecimiento de aprendizaje profundo, se volvieron a explorar las conexiones entre el aprendizaje de máquina y la computación. En un inicio, fue explorada la pregunta de qué tanto las redes neuronales pueden emular componentes computacionales o en qué grado pueden ser usadas para hacer el cómputo más eficiente en problemas de interés.

En particular, se exploró cómo las redes neuronales pueden emular máquinas de Turing (Neural Turing Machines, [Graves et al., 2014]), implementar memoria similar a la memoria RAM (Memory Networks, [Weston et al., 2014]), hacer uso de colas y pilas para contar con capacidades de conteo y memoria ([Joulin and Mikolov, 2015]), aumentadas con un conjunto de operaciones aritméticas y lógicas básicas para responder preguntas como conteos condicionales([Neelakantan et al., 2016]) o con memoria y un codificador de lenguaje computacional para aprender a representar, interpretar

y ejecutar programas [Reed and de Freitas, 2016].

El modelo pix2code nace a partir de esta línea de investigación en donde se busca que las redes neuronales y el aprendizaje de máquina tengan aplicaciones al cómputo y desarrollo de software aprovechando de su capacidad de aprendizaje de grandes cantidades de datos.

Estos avances tienen el potencial para desembocar en aplicaciones que impactarían la tecnología y nuestro mundo de forma fundamental en los próximos años. El trabajo propuesto aquí, no extiende las capacidades de una red neuronal como algunos de los trabajos mencionados anteriormente. Sin embargo, cae en el area de programación inducida ya que explora el problema de generación de código utilizable.

1.2. Notación y software

Para legibilidad y consistencia, la siguiente notación se usa a través del trabajo. Para un entero $n \in \mathbb{N}$ definimos $[n] := \{1, \ldots, n\}$. Los escalares serán denotados con letras minúsculas $x \in \mathbb{R}$, los vectores con minúsculas y negritas $\mathbf{x} \in \mathbb{R}^n$ y las matrices y tensores con mayúsculas y negritas $\mathbf{X} \in \mathbb{R}^{m \times n}$. También, para denotar una submatriz dentro de una matriz se usarán secuencias de filas y columnas. Por ejemplo, $A_{i:i+q,j:j+p}$ indica la submatriz formada por las filas $[i, \ldots, i+q]$ y las columnas $[j, \ldots, j+p]$.

El lenguaje de programación utilizado para los ejercicios numéricos de este trabajo es *python* 3.6. Se usó la librería *tensorflow* para realizar diferenciación automática de los modelos. Para la arquitectura y entrenamiento de redes neuronales la librería *keras* y para manejo y transformación de datos

las librerías *h5py* y *numpy*.

El repositorio con el código usado puede encontrarse en la siguiente liga, https://github.com/DiegoAgher/pix2code.

1.3. Organización de la tesis

Este trabajo se organiza de la siguiente manera. En los Capítulos 2 y 3 se presenta una introducción a las técnicas de aprendizaje con redes neuronales y a modelos de arquitectura profundas. Las técnicas de aprendizaje son introducidas con un modelo lineal y gradualmente se llega a las arquitecturas usadas por el modelo *pix2code*.

El Capítulo 4 se hace la deconstrucción del modelo utilizado por [Beltramelli, 2017]. La arquitectura es una combinación de redes convolucionales y recurrentes que genera el código dada una interfaz gráfica.

Así mismo, el Capítulo 4 expone las dimensiones prácticas de la arquitectura pix2code como los supuestos básicos de regularidad en los datos que dan lugar al *dominio de lenguage específico* (DLE) usado para regularizar la expresividad del lenguaje. Este capítulo presenta una evaluación numérica de la arquitectura pix2code y describe los resultados de los experimentos numéricos en el trabajo original en [Beltramelli, 2017]. Así mismo, se describen arquitecturas alternativas a la original para la resolución del problema de generación de código. Una de ellas es una variación menos compleja y la otra una más compleja. La segunda alternativa hace uso de una técnica denominada como mecanismo de atención. Esta extensión se llevó a cabo con experimentos numéricos y se compara al modelo original haciendo uso

de las mismas métricas.

El Capítulo 5 hace una recapitulación de los temas expuestos en este trabajo, expone con qué contribuye a lectores iniciándose en aprendizaje de máquina y programación inducida, así como presentar algunas direcciones para trabajo futuro.

Capítulo 2

Aprendizaje de Máquina y Redes Neuronales

Ya que se han expuesto los distintos problemas y la gama de posibilidades para uso de las redes neuronales, este capítulo tiene la tarea de describir matemáticamente la construcción de estos algoritmos. Intuitivamente, las redes neuronales son funciones $f : \mathcal{X} \to \mathcal{Y}$ que transforman un espacio $\mathcal{X} \subset \mathbb{R}^p$, tratando de moldearlo para encontrar el mejor *encaje* de cierta variable objetivo en el espacio $\mathcal{Y} \subset \mathbb{R}^q$. En este capítulo se explicará qué recursos matemáticos utilizan y cómo lo hacen. Para construirlas de forma progresiva y describir cómo transforman datos, se utilizará como punto de partida una transformación lineal. Teniendo esta base, complicaremos el modelo a través de composición de funciones no lineales para después esbozar la arquitectura de una red neuronal básica, el *perceptron de múltiples capas*.

2.1. El caso lineal

Supongamos que se tiene un conjunto de datos $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=0}^{m} \subseteq \mathbb{R}^n \times \mathbb{R}$. Buscamos capturar el comportamiento de la relación $y_i \sim \mathbf{x}_i$ a través de un modelo lineal parametrizado por un vector $\boldsymbol{\beta} \in \mathbb{R}^n$, i.e. $y_i \sim h(\mathbf{x}_i|\beta)$. Supongamos entonces que $\mathcal{D}$ ha sido muestreado de una variedad lineal

$$\mathcal{H}(\beta) := \{(\mathbf{x}, y) \in \mathbb{R}^n \times \mathbb{R} : \mathbf{x} \cdot \boldsymbol{\beta}^T = y, \boldsymbol{\beta} \in \mathbb{R}^n\}. \tag{2.1}$$

El modelo (2.1) equivale a un modelo de regularización implícita ya que restringe el espacio de búsqueda al conjunto de funciones lineales. De esta manera, podemos estimar el modelo generador de datos buscando la función lineal que mejor se ajuste a los datos. Declarando

$$\mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix} \in \mathbb{R}^m, \qquad \mathbf{X} = \begin{bmatrix} \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_m \end{bmatrix} \in \mathbb{R}^{m \times n} \tag{2.2}$$

podemos formular el problema de estimación como un problema de optimización, es decir

$$\min_{\beta} \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|^2. \tag{2.3}$$

El problema (2.3) puede resolverse de manera analítica por medio de mínimos cuadrados asumiendo una serie de hipótesis estadísticas para modelar el error de muestreo en los datos. Suponiendo que $m \geq n$ y que $\text{rango}(\mathbf{X}) = n$, la solución analítica resulta en el estimador,

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{2.4}$$

Aunque la solución propuesta en (2.4) soluciona (2.3), no provee de un método para obtener soluciones al problema cuando el conjunto $\mathcal{D}$ es de gran tamaño ya que el costo de calcular (2.4) es $\mathcal{O}(n^3)$[1].

Para superar esta problemática, usamos un método numérico basado en el gradiente de (2.3). Para hacer esto, usamos la identidad

$$\|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|_2^2 = (\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^T(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) = \sum_{i=1}^{m}(y_i - \mathbf{x}_i \cdot \boldsymbol{\beta}^T)^2. \tag{2.5}$$

La expresión final en (2.5) es la función a optimizar variando el parámetro $\boldsymbol{\beta}$ para resolver (2.3). Definiendo la función predictiva de nuestro modelo como

$$h(\mathbf{x}_i|\boldsymbol{\beta}) = \mathbf{x}_i \cdot \boldsymbol{\beta}^T, \tag{2.6}$$

obtenemos que la pérdida es medida como el sesgo cuadrado entre la predicción y la medición. Definiendo $\hat{y}_i = h(\mathbf{x}_i|\boldsymbol{\beta})$, entonces,

$$l(\hat{y}_i, y_i) = (y_i - \hat{y}_i)^2 = (y_i - \mathbf{x}_i \cdot \boldsymbol{\beta}^T)^2. \tag{2.7}$$

Con esto re-escribimos (2.5) como

$$J(\boldsymbol{\beta}) := \sum_{i=1}^{m} l(\hat{y}_i, y_i) = \sum_{i=1}^{m}(y_i - \mathbf{x}_i \cdot \boldsymbol{\beta}^T)^2. \tag{2.8}$$

La ecuación (2.8) define la función de costo del problema. La función $J(\boldsymbol{\beta})$ es diferenciable por lo que el minimizador puede encontrarse por *el método*

[1] Costos computacionales del estimador $\hat{\beta}$

- $\mathbf{X}^T\mathbf{X}$: $\mathcal{O}(n^2m)$
- $\mathbf{X}^T\mathbf{y}$: $\mathcal{O}(nm)$
- $(\mathbf{X}^T\mathbf{X})^{-1}$: $\mathcal{O}(n^3)$

del gradiente, el cual dado el gradiente ∇J realiza la actualización

$$\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} - \alpha \nabla J. \tag{2.9}$$

El parámetro α es la *taza de aprendizaje* y sirve para controlar la magnitud del paso en la dirección de máximo descenso sobre la función $J(\boldsymbol{\beta})$. Calculado el gradiente de (2.8):

$$\begin{aligned} \nabla J(\boldsymbol{\beta}) &= \nabla \sum_{i=1}^{m} (y_i - \mathbf{x}_i \cdot \boldsymbol{\beta}^T)^2 \\ &= -2 \sum_{i=1}^{m} \mathbf{x}_i (y_i - \mathbf{x}_i \cdot \boldsymbol{\beta}^T) \end{aligned} \tag{2.10}$$

obtenemos la expresión para actualizar los parámetros

$$\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} + 2\alpha \sum_{i=1}^{m} \mathbf{x}_i (y_i - \mathbf{x}_i \cdot \boldsymbol{\beta}^T). \tag{2.11}$$

En aplicaciones reales, el tamaño de $\mathcal{D}$ suele ser muy grande por lo que usar la totalidad del conjunto de datos para actualizar el parámetro $\boldsymbol{\beta}$ en cada iteración puede ser muy costoso. Para superar esto, podemos estimar el gradiente con un subconjunto de los datos $\mathcal{D}_c \subset \mathcal{D}$, entonces,

$$\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} - 2\alpha \sum_{(\mathbf{x}_i, y_i) \in \mathcal{D}_c} \mathbf{x}_i (y_i - \mathbf{x}_i \cdot \boldsymbol{\beta}^T)^2. \tag{2.12}$$

El método descrito anteriormente resulta en el algoritmo del gradiente estocástico [Bottou et al., 2016] presentado en el Algoritmo 1. Este método actualiza los parámetros estimando el gradiente considerando $J_{c_k} := J(\mathcal{D}_{c_k}|\boldsymbol{\theta}_k) = \sum_{\mathcal{D}_{c_k}} l(h(\mathbf{x}_k|\boldsymbol{\theta}), y_k)$ con $\mathcal{D}_{c_k} \subset \mathcal{D}$ un subconjunto aleatorio. Entonces, los parámetros se actualizan de la siguiente manera,

$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \alpha J_{c_k}. \tag{2.13}$$

Algoritmo 1: Descenso Estocástico del Gradiente

1 inicializar: $\boldsymbol{\theta}$, α, k, itermax;
2 **while** *no convergencia* & $k < itermax$ **do**
3 Escoger aleatoriamente un subconjunto de observaciones $\mathcal{D}_{c_k} \subset \mathcal{D}$;
4 $\boldsymbol{\theta} = \boldsymbol{\theta} + \alpha - \nabla_{\boldsymbol{\theta}} J(\mathcal{D}_{c_k}, \boldsymbol{\theta})$;

En [Bottou et al., 2016] se prueba que si $\{J_{c_k}\}_{k=1}^{p}$ es una dirección de descenso en valor esperado, entonces $\boldsymbol{\theta}$ converge a un mínimo bajo un grupo de suposiciones que incluyen continuidad y diferenciabilidad tanto de J como de ∇J. Esto a pesar de que J_{c_k} puede no representar una dirección de descenso para J para algún c_k dado que es un proceso estocástico. Lo anterior supone que el proceso para obtener $\mathcal{D}_{c_k}$ es un muestreo uniforme independiente e idénticamente distribuido de $\mathcal{D}$. La prueba considera funciones tanto fuertemente convexas como no convexas así como α fijo o una secuencia decreciente, es decir

$$\{\alpha_k\}_{k=1}^{p} : \sum \alpha^2 < \text{ínf}\,. \tag{2.14}$$

2.2. Aproximación de soluciones a partir de modelos no lineales

Hemos expuesto cómo estimar soluciones a problemas de regresión a través del uso de gradiente estocástico. Sin embargo, en este primer acercamiento se consideró solamente modelos cuyas formas son lineales, i.e $h(\mathbf{x}|\beta) = \mathbf{x} \cdot \boldsymbol{\beta}$. Aunque estos pueden ser útiles para modelar conjuntos

de datos de baja complejidad, pueden ser insuficientes para conjuntos con mayor complejidad. No obstante, podemos seguir empleando Gradiente Estocástico para aproximar soluciones a problemas que usan modelos parametrizados no lineales, provisto que cumplen con las hipótesis de continuidad y diferenciabilidad que requiere el algoritmo. De manera general, dado un conjunto de datos,

$$\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=0}^{m} \subseteq \mathcal{X}^p \times \mathcal{Y}^q, \tag{2.15}$$

y una familia de modelos parametrizados,

$$\mathcal{H} = \{h(\mathbf{x}|\boldsymbol{\theta}) : h(\mathbf{x}|\boldsymbol{\theta}) \sim \mathbf{y}\}. \tag{2.16}$$

Se busca aquel que encuentra una generalización del fenómeno descrito por $\mathcal{D}$ minimizando el valor esperado, sobre un conjunto de datos, de una medida de riesgo o pérdida. Es decir, siendo l una función de pérdida encontrar $h(\mathbf{x}|\boldsymbol{\theta}) \in \mathcal{H}$ tal que,

$$\boldsymbol{\theta} = \arg\min_{\boldsymbol{\theta}} \mathbb{E}_{\mathcal{D}}[l(h(\mathbf{x}|\boldsymbol{\theta}), \mathbf{y})]. \tag{2.17}$$

A éste valor esperado se le denomina *riesgo esperado* y se define como

$$R(\boldsymbol{\theta}) = \mathbb{E}[l(h(\mathbf{x}|\boldsymbol{\theta}), \mathbf{y})]. \tag{2.18}$$

Para poder encontrar $\boldsymbol{\theta}$ que minimiza (2.18) se tendría que contar con completa información sobre la relación de $\mathbf{y}$ con $\mathbf{x}$. En la práctica, este no es el caso la mayoría de las veces. Es por esto que se busca minimizar el estimador del riesgo esperado,

$$J(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} l(h(\mathbf{x}_i|\boldsymbol{\theta}), \mathbf{y}_i). \tag{2.19}$$

A (2.19) se le conoce como *riesgo empírico o costo*. Notemos que en el caso expuesto en la sección anterior el riesgo empírico está definido por (2.8). Así, el problema de estimación puede ser formulado como un problema de optimización donde (2.19) es la función objetivo dado un espacio definido por los parámetros $\boldsymbol{\theta}$ y el conjunto de datos $\mathcal{D}$. Podemos expresar este problema como a continuación,

$$\boldsymbol{\theta}^* = \min_{\theta} J(\mathbf{y}, \mathbf{X}, \boldsymbol{\theta}). \tag{2.20}$$

2.3. Modelo lineal para clasificación

Existen conjuntos de datos en los que se busca modelar una variable categórica, es decir $y_i \in [K]$. Para abordar el problema de modelaje de este tipo de variables se formula un problema de estimación de probabilidad de pertenencia dada una condicionalidad de características de la observación x_i. Comenzamos con un caso simple en el cuál se cuentan con solo dos clases distintas.

2.3.1. Clasificación binaria

En esta sección replicamos la metodología empleada en la sección anterior formulando un modelo distinto capaz de abordar el caso $y_i \in \{0, 1\}$. Replanteamos el conjunto de datos para tener una respuesta binaria,

$$\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=0}^{m} \subseteq \mathbb{R}^n \times \{0, 1\}. \tag{2.21}$$

Para aproximar el modelo generador, se propone un modelo cuyo propósito sea modelar la probabilidad de pertenencia a una clase. Es decir su resultado

se restringirá al conjunto $[0, 1]$. El modelo propuesto en la Sección 2.1 no es adecuado para este problema dado que mapea los datos al espacio $\mathbb{R}$. Para corregir esto, consideremos la función sigmoide, $\sigma : \mathbb{R} \to [0, 1]$ definida por

$$\sigma(z) = \frac{1}{1 + \exp(-z)}. \tag{2.22}$$

Haciendo uso de esta función se propone el modelo lineal para clasificación,

$$h(\mathbf{x}|\boldsymbol{\theta}) = \sigma(\mathbf{x} \cdot \boldsymbol{\theta}), \tag{2.23}$$

el cual puede ser interpretado como la probabilidad de pertenencia a la clase uno y obtener por complementariedad la probabilidad de pertenencia a la clase cero. Esto es,

$$p = p(y = 1|\mathbf{x}) = h(\mathbf{x}|\boldsymbol{\theta}), \quad p(y = 0|\mathbf{x}) = 1 - p. \tag{2.24}$$

De esta manera la clase predicha por el modelo es aquella que maximiza la probabilidad de pertenencia, digase aquel que soluciona

$$\hat{y} = \underset{k \in \{0,1\}}{\arg\max}\, p(y = k|\mathbf{x}). \tag{2.25}$$

Podríamos pensar en usar el error de clasificación como función de pérdida, $l(\hat{y}, y) = \mathbf{1}\{y \neq \hat{y}\}$ siendo $\mathbf{1}\{\cdot\}$ la función indicadora. Sin embargo, usar esta función para medir la pérdida de cada predicción no es una elección adecuada por dos razones: (1) no refleja la seguridad del modelo al predecir cierta clase y (2) no es continua y por lo tanto tampoco es diferenciable. Esto impide que se pueda usar Gradiente Estocástico para minimizar la función de costo y así encontrar el modelo que mejor ajuste los datos. Para continuar usando la metodología descrita anteriormente proponemos

la siguiente función de pérdida para calificar la calidad de las predicciones, i.e probabilidades de pertenencia,

$$e(p, y) = -y \log(p) - (1 - y) \log(1 - p) \tag{2.26}$$

recordando que $p = p(y = 1) = h(\mathbf{x}|\theta)$. Esta función es llamada *entropía cruzada* y tiene dos características deseables: continuidad y diferenciabilidad. Así, podemos minimizarla usando Gradiente Estocástico. Ademas, notemos que su valor aumenta cuando la probabilidad p se aleja de la clase real lo cual refleja un comportamiento esperado para una función de pérdida. Si usamos (2.26) para calificar el modelo, se tiene la siguiente pérdida para cada observación,

$$l(h(\mathbf{x}_i|\theta), y_i) = -(y_i \log(h(\mathbf{x}_i|\boldsymbol{\theta})) + (1 - y_i) \log(1 - h(\mathbf{x}_i|\boldsymbol{\theta}))), \tag{2.27}$$

por lo que la función de costo se define usando todas las observaciones,

$$J(\boldsymbol{\theta}) = -\sum_{i=1}^{m} y_i \log(h(\mathbf{x}_i|\boldsymbol{\theta})) + (1 - y_i) \log(1 - h(\mathbf{x}_i|\boldsymbol{\theta})). \tag{2.28}$$

Con (2.23) como propuesta de modelo y (2.28) función de costo, se puede plantear el problema de optimización y usar Gradiente Estocástico para encontrar una solución considerando el gradiente de (2.28),

$$\nabla J(\boldsymbol{\theta}) = \sum_{i=1}^{m} \mathbf{x}_i (h(\mathbf{x}_i|\boldsymbol{\theta}) - y_i). \tag{2.29}$$

A este modelo se le conoce como regresión logística.

2.3.2. Clasificación Multinomial

Para generalizar el modelo anterior considerando que la variable respuesta pueda tomar más de dos valores, se reformula el conjunto de datos

de nuevo como,

$$\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=0}^{m} \subseteq \mathbb{R}^n \times [K]. \tag{2.30}$$

El modelo (2.23) estimaba $p(y = 1)$ proveyendo $p(y = 0)$ por complementariedad dado que se necesita cumplir $p(y = 1) + p(y = 0) = 1$. Se propone el siguiente modelo para el caso multinomial,

$$p(y = j|\mathbf{x}) = h(\mathbf{x}|\boldsymbol{\theta}_j) = \frac{\exp(\mathbf{x} \cdot \boldsymbol{\theta}_j)}{\sum_{j=1}^{K} \exp(\mathbf{x} \cdot \boldsymbol{\theta}_j)} \tag{2.31}$$

con $j \in [K]$. Con una propuesta de modelo predictivo hace falta escoger una función de pérdida para usar Gradiente Estocástico como en la Sección 2.1. La función de pérdida para clasificación multinomial es una generalización de la función propuesta en el caso binario,

$$l(h(\mathbf{x}_i|\boldsymbol{\theta}_j), y_i) = \sum_{j=1}^{K} \mathbf{1}\{y = j\} \log \frac{\exp(\mathbf{x} \cdot \boldsymbol{\theta}_j)}{\sum_{j'=1}^{K} \exp(\mathbf{x} \cdot \boldsymbol{\theta}_{j'})}. \tag{2.32}$$

Definiendo $\boldsymbol{\theta} = [\boldsymbol{\theta}_1, ..., \boldsymbol{\theta}_K]$, entonces la función de costo esta dada por

$$J(\boldsymbol{\Theta}) = -\sum_{i=1}^{m} \sum_{j=1}^{K} \mathbf{1}\{y_i = j\} \log \frac{\exp(\mathbf{x}_i \cdot \boldsymbol{\theta}_j)}{\sum_{j'=1}^{K} \exp(\mathbf{x}_i \cdot \boldsymbol{\theta}_{j'})}. \tag{2.33}$$

Para minimizar esta función podemos usar también Gradiente Estocástico al igual que los problemas expuestos anteriormente. Similarmente al caso binomial, la predicción para evaluar métricas discretas, como precisión, es la clase con mayor probabilidad, es decir $\hat{y} = \arg\max_j p(y = j|\mathbf{x})$. A este modelo se le conoce como regresión logística multinomial o *regresión softmax*, en adelante nos referiremos a ella solamente como softmax.

Los casos expuestos anteriormente, regresión y clasificación, comparten una característica: se trata de inferir una función h a partir de un conjunto

de datos etiquetados $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{m} \subseteq \mathbb{R}^n \times \mathbb{R}^n$ o $\subset \mathbb{R}^n \times \mathbb{N}^n$ tal que $\mathbf{y} \sim h(\mathbf{x})$, donde $\mathbf{y}_i$ es una etiqueta o variable a modelar de la observación $\mathbf{x}_i$. A este tipo de problemas se les denomina problemas de *aprendizaje supervisado*.

2.4. Aproximación de funciones no lineales

En las secciones pasadas se expuso cómo usando modelos lineales se pueden aproximar soluciones a problemas tanto de regresión como de clasificación. En esta sección analizamos la función XOR para motivar los modelos no lineales. La función XOR:$\{0, 1\} \times \{0, 1\} \to \{0, 1\}$ está definida por la relación presentada en el Cuadro 2.1.

x_1	x_2	XOR(x_1, x_2)
0	0	0
0	1	1
1	0	1
1	1	0

Cuadro 2.1: Función XOR

No es posible que un modelo de clasificación lineal aproxime de forma correcta esta función ya que no se puede trazar una línea recta para separar las distintas clases. Para ver esto, supongamos que queremos aproximar ésta función con otra función continua. Usando x_1 y x_2 como coordenadas en el plano cartesiano se puede graficar la función XOR en la Figura 2.1.

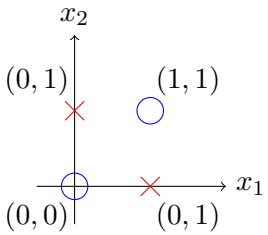


Figura 2.1: Función XOR

Sea $\mathbf{x} = [1, x_1, x_2]$ y $\boldsymbol{\beta} = [\beta_0, \beta_1, \beta_2]$. Supongamos que queremos construir un modelo lineal para clasificación como expuesto en la Sección 2.3.1,

$$h(\mathbf{x}|\beta) = \sigma(\mathbf{x} \cdot \beta^T). \tag{2.34}$$

Definiendo $p(y = 1|\mathbf{x}) = h(\mathbf{x}|\beta)$ entonces la predicción discreta será $\hat{y} = \arg\max_{j \in \{0,1\}} p(y = j|\mathbf{x})$ o equivalentemente,

$$\hat{y} = \begin{cases} 1 & h(\mathbf{x}|\boldsymbol{\beta}) \geq \frac{1}{2} \\ 0 & e.o.c. \end{cases}$$

Se sabe que,

$$\sigma(0) = \frac{1}{1 + \exp(-0)} = \frac{1}{2} \implies \sigma(z) \geq \frac{1}{2}, \forall z \geq 0. \tag{2.35}$$

Es decir, $z(\mathbf{x}) = \mathbf{x} \cdot \beta^T = \beta_0 + x_1 \cdot \beta_1 + x_2 \cdot \beta_2 \geq 0 \implies \hat{y} = 1$. En otras palabras, la clasificación depende de la separación de $\mathbb{R} \times \mathbb{R}$ inducida por el plano definido por $\mathbf{x} \cdot \beta^T$. Al separar en dos el espacio $\mathbb{R} \times \mathbb{R}$, un modelo de este tipo induce dos conjuntos,

$$G_1 := \{(x_1, x_2) \in \mathbb{R} \times \mathbb{R} : \mathbf{x} \cdot \beta^T \geq 0\}, \tag{2.36}$$

y el conjunto complemento,

$$G_2 := \mathbb{R} \times \mathbb{R} - G1 = \{(x_1, x_2) \in \mathbb{R} \times \mathbb{R} : \mathbf{x} \cdot \beta^T < 0\}. \tag{2.37}$$

Aunque se puede usar este modelo para pares $(x_1, x_2) \in \mathbb{R} \times \mathbb{R}$, en realidad solo es de interés evaluarlo en el espacio $\{0, 1\} \times \{0, 1\}$ para aproximar la función XOR. Con esto en consideración, G_1 y G_2 serán conjuntos finitos y existen solamente tres casos resultantes de un modelo que estima al generador,

1. $|G_1| = 0 \implies |G_2| = 4$ o $|G_1| = 4 \implies |G_2| = 0$
2. $|G_1| = 1 \implies |G_2| = 3$ o $|G_1| = 3 \implies |G_2| = 1$
3. $|G_1| = 2 \implies |G_2| = 2$.

Es decir, en el caso 1 los cuatro posibles resultados de la función a modelar pertenecen a G_1 o G_2. Este caso se visualiza en la Figura 2.2.

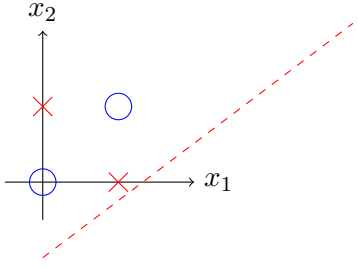


Figura 2.2: Caso $|G_1| = 0$

Claramente, no se buscan modelos que caigan en este caso ya que serían una mala aproximación. Para un modelo que aproxime correctamente esta

función se necesita caer en el caso 3 de forma que o $G_1 = \{(1,0),(0,1)\} \implies G_2 = \{(0,0),(1,1)\}$ o $G_2 = \{(1,0),(0,1)\} \implies G_1 = \{(0,0),(1,1)\}$. Veamos el siguiente diagrama en la Figura 2.3 para explorar los posibles modelos que caen en el caso 2, modelos aun incapaces de aproximar correctamente la función generadora. Después de explorar los modelos que caen en el caso 3 habremos probado por exhaustividad que no existe modelo lineal que aproxime de manera correcta la función a modelar.

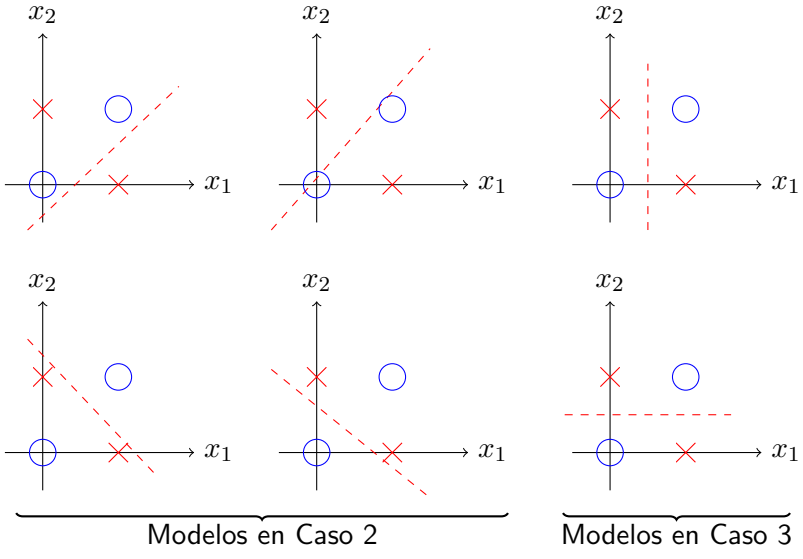


Figura 2.3: Casos de clasificadores

Una vez expuesto los casos posibles, concluimos que no hay un modelo lineal que aproxime esta función de forma correcta. Sin embargo, observando la Figura 2.1 podemos pensar en que para separar las clases dadas por la función XOR, cruces y círculo, se necesita un modelo no lineal capaz de *doblar* el espacio para luego clasificar linealmente. Proponemos una composición de funciones no lineales para lograr este *doblaje* del espacio. Sea

$h^{(2)}$ nuestro modelo, si $\mathbf{x} = [x_1, x_2]$ entonces,

$$\begin{aligned} a^{(1)}(\mathbf{x}) &= \mathbf{b}^{(1)} + \mathbf{W}^{(1)} \cdot \mathbf{x}, & h^{(1)}(\mathbf{x}) &= \sigma(a^{(1)}(\mathbf{x})) \\ a^{(2)}(\mathbf{x}) &= \mathbf{b}^{(2)} + \mathbf{W}^{(2)} \cdot h^{(1)}(\mathbf{x}), & h^{(2)}(\mathbf{x}) &= \sigma(a^{(2)}(\mathbf{x})) \end{aligned} \tag{2.38}$$

con los parámetros del modelos,

$$\begin{aligned} \mathbf{W}^{(1)} &= \begin{bmatrix} 100 & 100 \\ -100 & -100 \end{bmatrix}, & \mathbf{b}^{(1)} &= [-90, 110] \\ \mathbf{W}^{(2)} &= \begin{bmatrix} 100 & 100 \end{bmatrix} & \mathbf{b}^{(2)} &= [-110]. \end{aligned} \tag{2.39}$$

En el Cuadro 2.2 se analiza el comportamiento de este modelo caso por caso considerando los posibles valores de entrada en ambos argumentos de la función XOR para comprobar que esta es aproximada correctamente.

x_1	x_2	$h^{(1)} \approx$	$h^{(2)} \approx$
0	0	$[0, 1]$	0
0	1	$[1, 1]$	1
1	0	$[1, 1]$	1
1	1	$[1, 0]$	0

Cuadro 2.2: Casos resultantes de (2.38)

Dados estos resultados, vemos que este modelo logra aproximar la función XOR. Observemos $h^{(1)}$ por componentes del vector resultante. La primera mapea el espacio de entrada como una disyunción lógica,

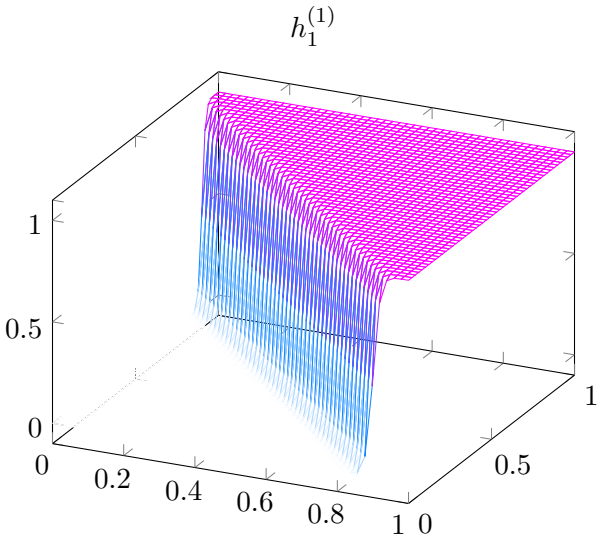


Figura 2.4: Mapeo producido por neurona $h_1^{(1)}$

Por otro lado, la segunda actúa como reflexión de una disyunción, si al menos una entrada del vector es cero, se tiene un resultado mayor a cero.

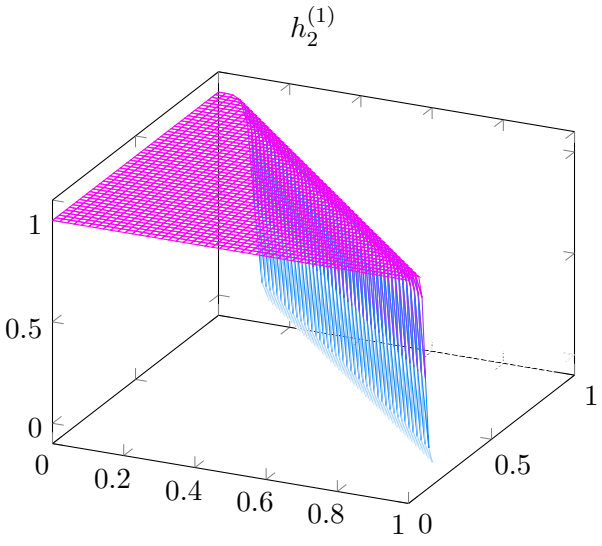


Figura 2.5: Mapeo producido por neurona $h_2^{(1)}$

Ahora, $h^{(2)}$ actúa sobre estos dos operadores anteriores como si fuera una conjunción lógica logrando encontrar el *encaje* apropiado para separar las clases de la función XOR.

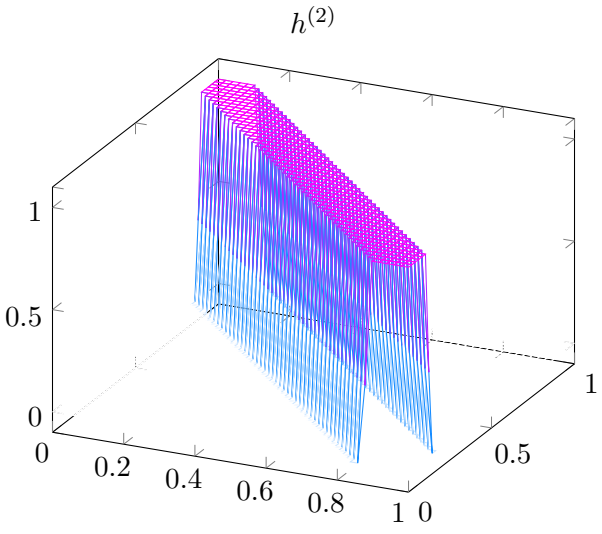


Figura 2.6: Mapeo producido por $h^{(2)}$

Con este ejemplo podemos observar cómo el uso de composiciones de funciones no lineales puede encontrar un modelo clasificador, en un nuevo espacio, para conjuntos de datos no linealmente separables. Con este ejemplo como intuición, la siguiente sección generaliza la construcción y entrenamiento de modelos como este basados en la idea de composicionalidad de funciones no lineales.

2.5. El Perceptrón de Múltiples Capas

El conjunto de ecuaciones (2.38) propuesto en la sección anterior puede ser representado con un grafo. Matemáticamente, si N es un conjunto finito, un grafo es una tupla $G = (N, A)$ con $A \subset N \times N$. Los grafos son particularmente útiles para la representación y modelaje de flujos a través

de estructuras con cierta conectividad. Esto es relevante en nuestro caso ya que la transformación de datos a través del conjunto de ecuaciones (2.38) puede ser modelada a través de la composicionalidad inducida por las funciones de este sistema. Es decir, si redefinimos $\mathbf{x} = [x_1, x_2, 1]$ podemos reescribir (2.38) como

$$\begin{aligned} a^{(1)}(\mathbf{x}) &= \mathbf{W}^{(1)} \cdot \mathbf{x}, & h^{(1)}(\mathbf{x}) &= \sigma(a^{(1)}(\mathbf{x})) \\ a^{(2)}(\mathbf{x}) &= \mathbf{W}^{(2)} \cdot [h^{(1)}(\mathbf{x}), 1] & h^{(2)}(\mathbf{x}) &= \sigma(a^{(2)}(\mathbf{x})), \\ \mathbf{W}^{(1)} &= \begin{bmatrix} 100 & 100 & -90 \\ -100 & -100 & 110 \end{bmatrix}, & \mathbf{W}^{(2)} &= \begin{bmatrix} 100 & 100 & -110 \end{bmatrix}. \end{aligned} \tag{2.40}$$

De esta manera, podemos representar el flujo y transformación de datos en (2.38) a través del grafo con conjunto de nodos $N = \{\mathbf{x}, h^{(1)}(\mathbf{x}), h^{(2)}(\mathbf{x})\}$ y conjunto de aristas dado por $A = \{\mathbf{W}^{(1)}, \mathbf{W}^{(2)}\}$. Mostramos este grafo en la Figura 2.7.

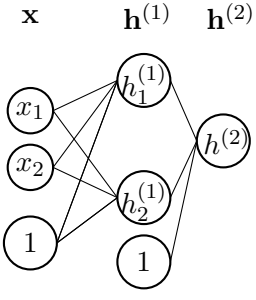


Figura 2.7: Grafo representando (2.40)

Los grafos son una manera de representar la composicionalidad entre funciones de diferentes clases. El ejemplo anterior (correspondiente a (2.40)) muestra la representación de una red de composicionalidad con tres capas,

que corresponden a la capa de entrada $\mathbf{x}$, la capa escondida $\mathbf{h}^{(1)}$ y la capa de salida $\mathbf{h}^{(2)}$.

El modelo presentado en (2.40) es un caso del modelo conocido como perceptrón de *una* capa escondida. Este puede ser generalizado a L capas escondidas dando lugar a un modelo con mayor capacidad de aprendizaje. De manera general, si g es un funcional no lineal y $\{\mathbf{W}^{(l)} \in \mathbb{R}^{n_l \times (n_{l-1}+1)} : l \in [L+1]\}$ es un conjunto de matrices y $n_l \in \mathbb{N}, \forall l \in [L+1]$, el perceptrón con L capas puede ser definido de manera recurrente por

$$\begin{aligned} h^{(0)}(x) &= [\mathbf{x}, 1] \\ a^{(l)}(\mathbf{x}) &= \mathbf{W}^{(l)} \cdot [h^{(l-1)}(\mathbf{x}), 1] \\ h^{(l)}(x) &= g(a^{(l)}(\mathbf{x})). \end{aligned} \tag{2.41}$$

El grafo asociado a (2.41) está definido por

$$V = \{h^{(l)}(\mathbf{x})\}_{l=0}^{L+1}, A = \{\mathbf{W}^{(l)}\}_{l=0}^{L+1}. \tag{2.42}$$

El modelo presentado en (2.41) y perceptrón de una capa escondida son dos arquitecturas de la familia de modelos parametrizados llamados redes neuronales. En el resto del capítulo, elaboramos de manera detallada la construcción matemática de la arquitectura del perceptrón de múltiples capas.

Se comienza dicha descripción por el bloque unitario de estos modelos, las neuronas. En general, una neurona es una función $f : \mathbb{R}^m \to \mathbb{R}$. En la formulación (2.41) una neurona está definida por,

$$h_j^{(l)}(\mathbf{x}) : \mathbb{R}^{n_{l-1}} \to \mathbb{R} \tag{2.43}$$

con $j \in [n_{l-1}]$. Las neuronas son los nodos en las redes neuronales y construyen representaciones no lineales de su entrada. Explícitamente, las neuronas son una composición de funciones,

$$a^{(l)}(\mathbf{x})_j = \mathbf{W}^{(l)}_{\cdot j} \cdot [\mathbf{h}^{(l-1)}(\mathbf{x}), 1], \quad h^{(l)}(a(\mathbf{x}))_j = g(a(\mathbf{x})_j), \tag{2.44}$$

donde $a^{(l)}(\mathbf{x})_j$ se denomina la preactivación de la neurona y g la función de activación. Consecuentemente nos referimos a $h^{(l)}(a(\mathbf{x}))_j$ como la activación de la neurona. En (2.40), por ejemplo, la función de activación es la función sigmoide, i.e. $g = \sigma$.

La función de activación, g, es una función no lineal que determina el *disparo* o activación de la neurona. En la practica, se usan distintas funciones de activación y su elección se toma dependiendo del problema a resolver. La Tabla 2.3 muestra algunas funciones de activación usadas comúnmente así como su expresión y el espacio al que mapea sus argumentos.

Nombre	Expresión matemática	Imagen
Lineal	$g(a) = a$	$(-\text{ínf}, \text{ínf})$
Sigmoide	$g(a) = \sigma(a)$	$[0, 1]$
Tangente hiperbólica	$g(a) = \frac{\exp(a)-\exp(-a)}{\exp(a)+\exp(-a)}$	$[-1, 1]$
Lineal Rectificada (RELU)	$g(a) = \text{máx}(0, a)$	$[0, \text{ínf})$

Cuadro 2.3: Funciones de activación

Se denomina una capa de la red (o capa oculta) al vector de neuronas que comparten el argumento de entrada. De manera formal, siguiendo (2.41) una capa se expresa como

$$h^{(l)}(\mathbf{x}) : \mathbb{R}^{n_{l-1}} \to \mathbb{R}^{n_l} \tag{2.45}$$

En el perceptrón de capas múltiples presentado en (2.41) existen 3 tipos de capas: datos, capas ocultas y capa de salida. En la Tabla 2.4 se presentan las definiciones de cada una.

Capa	Definición
De entrada o datos	$h^{(0)}((x)) = \mathbf{x}$
Oculta o densa	$h^{(l)}((x)) : l \in [L]$
De salida	$h^{(L+1)}((x))$

Cuadro 2.4: Tipos de Capas

Ahora, la formulación de $h^{(L+1)}$, llamada capa de salida, de hecho depende del problema a resolver. En las secciones 2.1 y 2.3 se expuso respectivamente, el problema de regresión y clasificación, utilizamos estos para definir la capa de salida de una red neuronal.

- Regresión: Estimación lineal sobre la última capa.

$$y \in \mathbb{R}^n \implies h^{(L+1)}(\mathbf{x}) = a^{(L+1)}(\mathbf{x}) \tag{2.46}$$

- Clasificación: Regresión softmax sobre la última capa.

$$y \in [K]^n \implies h^{(L+1)}(\mathbf{x}) = \text{softmax}(a^{(L+1)}(\mathbf{x})) \tag{2.47}$$

Para simplificar notación, definimos $\mathbf{h}^{(k)} := h^{(k)}(\mathbf{x})$, observamos en (2.41) que $h_j^{(k)}$ depende solamente de $\mathbf{h}^{(k-1)}$ más no de $h_i^{(k)}, \forall i \neq j$. Así, este modelo puede pensarse como una gráfica dirigida acíclica que representa flujo de información. La información inicial, los datos de entrada, fluye a través de las capas ocultas transformándose hasta llegar a la capa de salida.

Con esta formulación de red neuronal, visualizamos el grafo asociado en la Figura 2.8.

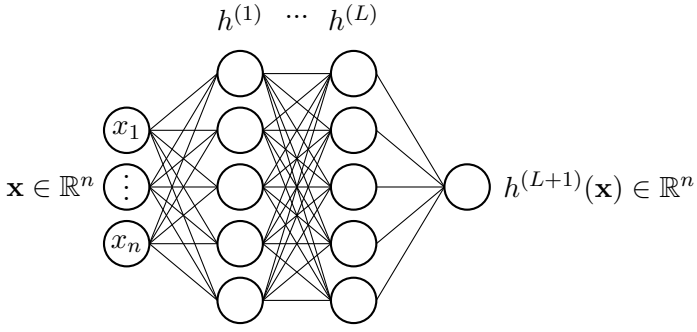


Figura 2.8: Grafo de Red Neuronal de regresión

El modelo descrito en (2.41) y el perceptrón de una capa no son los únicos modelos o *arquitecturas* de redes neuronales. En el Capítulo 3 expondremos dos arquitecturas, distintas a la expuesta en esta sección, para modelar de forma más eficiente conjuntos de datos con distintas naturalezas.

2.6. Entrenamiento de redes neuronales

Hasta ahora se ha descrito la construcción de una arquitectura de redes neuronales para resolver problemas de regresión o clasificación. Sin embargo, no se ha detallado el método para aproximar soluciones que permitan modelar los datos de forma correcta. Para usar Gradiente Estocástico, como se hizo en las secciones anteriores, se necesita proponer una función de pérdida para plantear un problema de optimización y así encontrar una

aproximación a la solución. Las funciones de pérdida necesitan ser continuas y diferenciables para que el modelo también lo sea de principio a fin y sea posible usar Gradiente Estocástico. Recordando que se busca encontrar el mínimo de la función de costo o pérdida para encontrar los parámetros del modelo, usando Gradiente Estocástico, buscamos estimar ∇J_Θ siendo J la función de costo del problema de optimización y Θ el conjunto de parámetro del modelo. La dificultad principal de usar este método es la estimación del gradiente. A continuación se presenta una forma de estimarlo con un ejemplo de un perceptrón de múltiples capas para un problema de regresión.

2.6.1. Gradiente de una red neuronal

Consideremos un modelo descrito por (2.41) con L capas escondidas para un problema de regresión. La capa de salida $L+1$ está definida como $h^{(L+1)}(\mathbf{x}) = \mathbf{W}^{(L+1)} \cdot [h^{(L)}(\mathbf{x}), 1]$ y como función de pérdida el error cuadrático. Sea $\hat{y} = h^{(L+1)}(\mathbf{x})$ entonces,

$$l(\hat{y}_i, y_i) = (h^{(L+1)}(\mathbf{x}_i) - y_i)^2. \tag{2.48}$$

Se usa Gradiente Estocástico para resolver el problema de minimización,

$$\min_{\Theta} J(\Theta) = \min_{\Theta} \sum_{i=1}^{m} l(h^{(L+1)}(\mathbf{x}_i, \Theta), y_i)), \tag{2.49}$$

y encontrar una aproximación a la solución. En el siguiente desarrollo se hace un abuso de notación para denotar gradientes. Se utiliza la notación de derivadas parciales para denotar gradiente de la función de costo con respecto a alguna preactivación o activación del modelo. De esta forma,

las expresiones para el gradiente son legibles y concisas. Por ejemplo, la expresión $\frac{\partial f}{\partial \mathbf{a}^{(l)}}$ hace referencia al gradiente de f con respecto al vector $\mathbf{a}^{(l)}$.

Para calcular $\frac{\partial l(\hat{y}_i, y_i)}{\partial \mathbf{W}^{L+1}}$ solo necesitamos recurrir a las definiciones. Sea $l_i := l(\hat{y}_i, y_i)$ entonces,

$$\begin{aligned} \frac{\partial l_i}{\partial \mathbf{W}^{L+1}} &= \frac{\partial (\mathbf{W}^{(L+1)} \cdot h^{(L)}(\mathbf{x}_i) - y_i)^2}{\partial \mathbf{W}^{(L+1)}} \\ &= 2(\mathbf{W}^{(L+1)} \cdot h^{(L)}(\mathbf{x}_i) - y_i) h^{(L)}(\mathbf{x}_i). \end{aligned} \tag{2.50}$$

Nótese la jerarquía de las capas escondidas $h^{(l)}(\mathbf{x})$. Por lo cual, para el gradiente de los parámetros del resto de las capas, $\forall l \leq L$, se debe usar regla de la cadena obteniendo la siguiente expresión,

$$\frac{\partial l_i}{\partial \mathbf{h}^{(l)}} = \frac{\partial l_i}{\partial \mathbf{h}^{(L+1)}} \cdot \prod_{j=l}^{L} \frac{\partial \mathbf{h}^{(j+1)}}{\partial \mathbf{h}^{(j)}}. \tag{2.51}$$

Por lo tanto la expresión para el gradiente de los parámetros de cada capa será,

$$\frac{\partial l_i}{\partial \mathbf{W}^{(l)}} = \frac{\partial l_i}{\partial \mathbf{h}^{(l)}} \cdot \frac{\partial \mathbf{h}^{(l)}}{\partial \mathbf{W}^{(l)}}. \tag{2.52}$$

Para contar con una expresión para el gradiente de los parámetros en cada capa escondida definimos el gradiente de estos con respecto a la activación de la capa,

$$\frac{\partial \mathbf{h}^{(l)}}{\partial \mathbf{W}^{(l)}} = \frac{\partial g(\mathbf{W}^{(l)} \cdot \mathbf{h}^{(l-1)})}{\partial \mathbf{W}^{(l)}} = g'(\mathbf{W}^{(l)} \cdot \mathbf{h}^{(l-1)})(\mathbf{h}^{(l-1)})^T, \tag{2.53}$$

con $g'(\cdot)$ la derivada de la función de activación elemento a elemento.

Con la derivación de (2.51) estamos listos para describir el algoritmo de *propagación hacia atrás*, elemento clave para el entrenamiento de redes neuronales. Durante Gradiente Estocástico, la estimación de $\frac{\partial l_i}{\partial \mathbf{W}^{(l)}}$ usando

(2.51) depende del gradiente de toda capa superior a esta, de ahí el nombre del algorítmo. Este, describe el flujo del gradiente del modelo desde la capa de salida hasta la primera capa escondida. Al flujo de información empezando por la capa de entrada, a través de las capas ocultas, hasta la capa de salida se le denomina *paso hacia delante*.

Durante entrenamiento, no es raro que se encuentren mínimos locales en vez de absolutos que generalizan el comportamiento de los datos. Para asegurar no caer en puntos locales, se necesita hacer otro tipo de análisis sobre la matriz *Hessiana*, es decir la matriz de segundas derivadas. El cálculo de la matriz Hessiana para estos modelos es costoso y crece a medida que se aumenten capas en él. Al usar Gradiente Estocástico para la optimización del modelo se tiene la ventaja de poder usar el criterio de paro temprano. Este criterio consta de dejar de entrenar el modelo al detectar que una métrica de desempeño sobre el *conjunto de validación*, en vez de mejorar, empeora. El conjunto de validación es un conjunto de datos del fenómeno a modelar que no se incluye durante entrenamiento, es decir el modelo jamás ha visto estas observaciones. Esta estrategia de paro temprano permite al modelo seguir encontrando distintos mínimos a pesar de haberse encontrado con uno local. También es usada para evitar sobreajustes a los datos de entrenamiento.

A continuación presentamos una manera de encontrar gradientes en un grafo computacional usando el paso hacia atrás, la forma de optimizar modelos gráficos programáticamente.

2.6.2. Propagación hacia atrás

El perceptrón de múltiples capas es un modelo con una estructura clara, en el cual los gradientes pueden ser derivados de manera analítica. Con otro tipo de arquitecturas más complicadas esto no es posible por lo que se requiere de una metodología automatizada. Para esto, se usan técnicas de diferenciación automática [Schulman et al., 2015] para entrenar distintas arquitecturas de forma programática. Pensemos estos modelos como grafos computacionales dirigidos. Es decir, en un grafo los nodos representarán operaciones a efectuar, y valores del resultado de estas, usando como argumentos valores provenientes de otros nodos a través de aristas que llegan a él. Dado que estas estructuras reflejan composición de funciones en esta representación de una función, usaremos dos elementos para calcular el gradiente de esta:

- El flujo de información a través del grafo.
- La regla de la cadena, si $z = f(x, y), x = h(t), y = g(t) \implies \frac{\partial z}{\partial t} = \frac{\partial z}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial z}{\partial y}\frac{\partial y}{\partial t}$.

Consideremos el siguiente ejemplo para describir el proceso de diferenciación automática. Sea

$$\mathbf{x} = [1, x_1, x_2], \quad \boldsymbol{\beta} = [\beta_0, \beta_1, \beta_2], \quad f(\mathbf{x}) = \sigma(\mathbf{x} \cdot \boldsymbol{\beta}^T) \tag{2.54}$$

Proponemos la refactorización,

$$q_1 = x_1 * \beta_1, \quad q_2 = x_2 * \beta_2, \quad q_3 = \beta_0 + q_1 + q_2 \quad \implies f(\mathbf{x}) = \sigma(q_3). \tag{2.55}$$

Utilizando (2.55) podemos descomponer f en un grafo computacional por funciones intermedias en el cual q_1, q_2, q_3 y σ son nodos. Este grafo puede ser visualizado en la Figura 2.9.

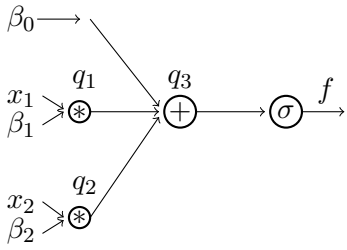


Figura 2.9: Grafo computacional de $f(\mathbf{x}) = \sigma(\mathbf{x}^T \cdot \beta)$

Dado que $f(x)$ puede ser expresada como una composición de funciones reflejada en el grafo, los nodos pueden calcular su gradiente local con respecto a sus argumentos y usar la regla de la cadena para calcular gradientes globales. Calculamos $\frac{\partial f}{\partial \boldsymbol{\beta}_1}$ primero usando regla de la cadena y luego a través del grafo. Sabemos que por regla de la cadena,

$$\frac{\partial f}{\partial \beta_1} = \frac{\partial \sigma(q_3)}{\partial q_3} \cdot \frac{\partial q_3}{\partial q_1} \cdot \frac{\partial q_1}{\partial \beta_1} \tag{2.56}$$

Esto es porque en el grafo, el gradiente local del nodo σ analíticamente corresponde a $\frac{\partial \sigma(q_3)}{\partial q_3} = \sigma'(q_3)$. Recorriendo los arcos hacia atrás hasta llegar a β_1 calculamos los gradientes locales de q_3,

$$\frac{\partial q_3}{\partial \beta_0} = 1, \quad \frac{\partial q_3}{\partial q_1} = 1, \quad \frac{\partial q_3}{\partial q_2} = 1. \tag{2.57}$$

Una vez más calculamos el gradiente local, esta vez para el nodo q_1:

$$\frac{\partial q_1}{\partial \beta_1} = x_1. \tag{2.58}$$

Así, el gradiente de f con respecto a β_1 se puede obtener a través del producto de los gradientes locales siguiendo los arcos del grafo:

$$\frac{\partial f(x)}{\partial \beta_1} = \frac{\partial \sigma'(q_3)}{\partial q_3} \cdot \frac{\partial q_3}{\partial q_1} \cdot \frac{\partial q_1}{\partial \beta_1} = \sigma'(q_3) \cdot 1 \cdot x_1. \tag{2.59}$$

Esta última expresión corresponde a la obtenida en (2.56) a través de la regla de la cadena. La Figura 2.10 ayuda a visualizar el recorrido del gradiente a través del grafo.

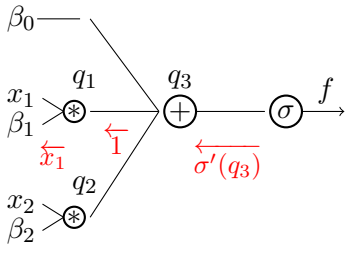


Figura 2.10: Gradientes locales para calcular $\frac{\partial f}{\partial \beta_1}$

El procedimiento para β_0 y β_2 sería el mismo considerando los distintos valores de los nodos. El paso hacia delante determina los valores de cada nodo en este sentido. Asignando valores para los parámetros y datos de entrada ejemplificamos el paso hacia delante en azul y el paso hacia atrás en rojo en la Figura 2.11.

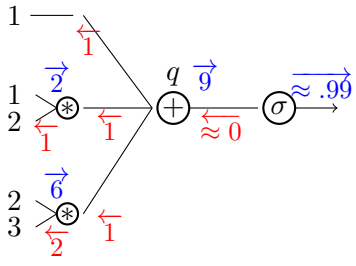


Figura 2.11: Gradientes locales con valores numéricos para calcular ∇f

En resumen, empezando por el nodo final de la red (función objetivo) calculamos el gradiente multiplicando el gradiente local de un nodo con todo otro que esté conectado a través de aristas hacia atrás hasta llegar a los nodos de los parámetros.

Usando este algoritmo se puede encontrar el gradiente de un grafo computacional tan compleja como una red neuronal de forma programable sin necesidad de una formulación analítica. En el grafo de una red neuronal las operaciones más frecuentes son la suma, multiplicación, máximo, función sigmoide. Analicemos el comportamiento de los gradientes locales para algunas operaciones presentes en este ejemplo,

operación	gradiente
$+$	$\frac{\partial a+b}{\partial a} = 1$
$*$	$\frac{\partial a*b}{\partial a} = b$
max	$\frac{max(a,b)}{\partial a} = \mathbf{1}_{max(a,b)}$

Cuadro 2.5: Operaciones y su forma de propagar el gradiente

- El operador suma, distribuye el gradiente de forma equitativa a sus

argumento dado que el gradiente local es unitario.

- El operador producto escala por un factor del otro argumento. Esto es importante a considerar debido a la común formulación de los productos punto: $\mathbf{x}^T \cdot \boldsymbol{\beta}$. El gradiente del parámetro $\boldsymbol{\beta}$ puede aumentar o disminuir en al menos la escala de $\mathbf{x}$. Intuitivamente se puede saber qué podría pasar con las magnitudes de los parámetros dadas las magnitudes de los datos.

- El operador máximo escala por un factor de su gradiente al nodo cuyo valor sea el mayor de sus entradas.

2.6.3. Problema del gradiente desvaneciente

Al entrenar redes profundas, la estimación del gradiente puede ser problematica debido a que este tiende a crecer/decrecer exponencialmente. Para analizar este fenómeno [Arjovsky et al., 2016] hace uso de la expresión para derivar el gradiente de una capa oculta de (2.51),

$$\frac{\partial l_i}{\partial \mathbf{h}^{(l)}} = \frac{\partial l_i}{\partial \mathbf{h}^{(L+1)}} \cdot \prod_{j=l}^{L} \frac{\partial \mathbf{h}^{(j+1)}}{\partial \mathbf{h}^{(j)}}. \tag{2.60}$$

Vemos que el producto de gradientes con respecto a la capa anterior tenderá a explotar o desvanecer a medida que L se haga más grande,

$$\left\| \frac{\partial l_i}{\partial \mathbf{h}^{(l)}} \right\| = \left\| \frac{\partial l_i}{\partial \mathbf{h}^{(L+1)}} \cdot \prod_{j=l}^{L} \frac{\partial \mathbf{h}^{(j+1)}}{\partial \mathbf{h}^{(j)}} \right\| = \left\| \frac{\partial l_i}{\partial \mathbf{h}^{(L+1)}} \cdot \prod_{j=l}^{L} g'(\mathbf{W}^{(j+1)}\mathbf{h}^{(k)})\mathbf{W}^{(j+1)} \right\|, \tag{2.61}$$

particularmente para las primeras capas del modelo donde en el producto tiene el mayor número de factores. Intuitivamente, el gradiente es el pro-

ducto de un número de factores que aumenta con el número de capas. Este, tenderá a cero si los factores son menores a uno o a infinito si son mayores a uno. La ecuación 2.61 hace a la función RELU una no linealidad interesante debido a su gradiente local,

$$\frac{\partial \,\mathrm{RELU}(x)}{\partial x} = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0. \end{cases}$$

Aún más, el producto de los parámetros $\mathbf{W}^{(l)}$ indica que la inicialización de estos es importante para evitar el problema. Supongamos $\mathrm{Var}(\mathbf{x}) = 1$, entonces $E[\mathbf{W}^{(l)}] = 0$ y $\mathrm{Var}(\mathbf{W}^{(l)}) = \frac{4}{n_{l-1}}$. Aunque existe la posibilidad de que un modelo con pocas capas no sufra de este problema, es una realidad que para modelos profundos, como los modelos recurrentes presentados en la Sección 3.3, se debe considerar para entrenarlos de forma exitosa[Bengio et al., 1994].

2.6.4. Regularización

Durante el periodo de entrenamiento existe el riesgo de *sobreajustar*. Eso es una situación en la cual el algoritmo ajusta los parámetros, durante la fase de entrenamiento, de tal forma que minimiza "a toda costa" la función de pérdida. Sin embargo, los datos de entrenamiento no representan la distribución total del fenómeno. Por esto, permitir al algoritmo minimizar la perdida dado ese conjunto de datos, probablemente encontrando un mínimo local, es encontrar el mejor modelo para *ese* conjunto de datos. He ahí el problema de sobreajustar: si se usa el modelo optimizado para los datos de entrenamiento, hacer inferencia sobre distintos datos resulta en métricas de

desempeño considerablemente más pobres que las de entrenamiento.

Una técnica popular para evitar este problema es agregar un término de penalización sobre los parámetros a la función a minimizar durante la etapa de entrenamiento,

$$J(\mathbf{x}|\boldsymbol{\theta}) = \mathcal{L}(\mathbf{x}, \boldsymbol{\theta}) + \gamma \|\boldsymbol{\theta}\|.$$

El parámetro γ es la penalización sobre el tamaño de los parámetros. Es por eso que a este método se le llama encogimiento ya que con $\gamma > 0$ el tamaño de los parámetros se penaliza. A manera que estos crecen, la función de costo también. Este tipo de regularización es conocida como *ridge* por su interpretación en inglés de penalizar mayormente variables que con "crestas" que aportan grandes contribuciones. Es decir, este tipo de regularización promueve modelos que usen muchas variables con contribuciones similares a diferencia de modelos ralos que usan pocas variables con fuertes contribuciones.

En [Hinton et al., 2012] se desarrolla una técnica de regularización para redes neuronales llamada *drop-out* comúnmente usada en perceptrones multicapas. Esta técnica consta de "tirar" las activaciones, junto con sus conexiones, de un subconjunto aleatorio de neuronas. Esto puede llevarse a cabo en los datos de entrada o en las capas escondidas. La motivación de esta técnica es que redes neuronales de gran tamaño, con mayor capacidad de aprendizaje, cuentan con grandes números de parámetros. Al tener grandes conjuntos de parámetros, datos complejos pueden ser ajustados a la perfección dando lugar al problema de sobreajustar. Esta situación sucede regularmente cuando el conjunto de entrenamiento es pequeño. En cada

caso de entrenamiento, las neuronas pueden o no estar con una probabilidad p, es decir, las neuronas no pueden depender totalmente de alguna otra(s) en específico. En el trabajo de [Hinton et al., 2012] tambien se prueba que esta técnica tiene mejores resultados en el conjunto de prueba del problema MNIST[2]. En este trabajo, se logra una mejora de alrededor de 15 % en el conjunto de validación. En conjunto con drop-out, también se hace una restricción de la norma 2 de los vectores de parámetros, esto es diferente a la penalización porque los vectores se normalizan en vez de penalizar la función costo.

[2]MNIST[LeCun and Cortes, 2010] es un conjunto de datos para algoritmos de aprendizaje estadístico e inteligencia artificial, contiene 60,000 imágenes tamaño 28x28 de dígitos trazados a mano. Estas son para entrenamiento y 10,000 más para prueba

Capítulo 3

Modelos para Imágenes y Texto

En el Capítulo 2 se presentó el perceptrón de capas múltiples, una arquitectura simple de redes neuronales. En este capítulo se hace uso de este modelo para aproximar una solución para el problema de clasificación de imágenes del conjunto MNIST. Como se mencionó en el capítulo anterior este conjunto consta de imágenes de dígitos de 28×28 trazados a mano. Tras explorar los resultados obtenidos, se analizan las limitantes de dicho modelo para manejar datos de imágenes cuya naturaleza es de cuadrícula. Con esta motivación, se propone una alternativa para el modelaje de este tipo de datos: la capa convolutiva. Después de exponer porqué el perceptrón multicapas tampoco es adecuado para modelar datos de texto con naturaleza secuencial, se termina el capítulo describiendo una arquitectura distinta para modelaje de este tipo de datos, las redes recurrentes. Adicio-

nalmente, se detalla el modelo LSTM [Hochreiter and Schmidhuber, 1997] por ser de particular relevancia a para este trabajo debido a su uso en [Beltramelli, 2017]. Estas dos capas, convolucional y LSTM, son la base del modelo *pix2code* presentado en el Capítulo 4.

3.1. Clasificación de images usando el perceptrón de capas múltiples

Dada la capacidad teórica del perceptrón (2.41) de resolver problemas de clasificación multinomial, se propone un modelo con tres capas escondidas y una capa de salida softmax para aproximar una solución al problema de clasificar observaciones del conjunto MNIST. Se define el conjunto de datos reacomodando las imágenes en un vector,

$$\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=0}^{m} \subseteq \mathbb{R}^{784} \times [10]. \tag{3.1}$$

Matemáticamente, el modelo para este problema se define de la siguiente manera,

$$\begin{aligned} a^{(1)}(\mathbf{x}) &= \mathbf{W}^{(1)} \cdot [\mathbf{x}, 1], & h^{(1)}(x) &= \text{RELU}^{\dagger}(a^{(1)}(\mathbf{x})) \\ a^{(2)}(\mathbf{x}) &= \mathbf{W}^{(2)} \cdot [h^{(1)}(\mathbf{x}), 1], & h^{(2)}(x) &= \text{RELU}^{\dagger}(a^{(2)}(\mathbf{x})) \\ a^{(3)}(\mathbf{x}) &= \mathbf{W}^{(3)} \cdot [h^{(2)}(\mathbf{x}), 1], & h^{(3)}(\mathbf{x}) &= \text{softmax}(a^{(3)}(\mathbf{x})), \end{aligned} \tag{3.2}$$

donde $\text{RELU}^{\dagger}$ significa que el resultado de la activación RELU fue procesada por una capa dropout con probabilidad de 50 %. Los parámetros del modelo son matrices,

$$\mathbf{W}^{(1)} \in \mathbb{R}^{n_1 \times (784+1)}, \quad \mathbf{W}^{(2)} \in \mathbb{R}^{n_2 \times (n_1+1)}, \quad \mathbf{W}^{(3)} \in \mathbb{R}^{10 \times (n_2+1)} \tag{3.3}$$

inicializadas aleatoriamente con $n_1 = 1000, n_2 = 2000$. La visualización del grafo asociado la red anterior ((3.2)) puede verse en la Figura 3.1.

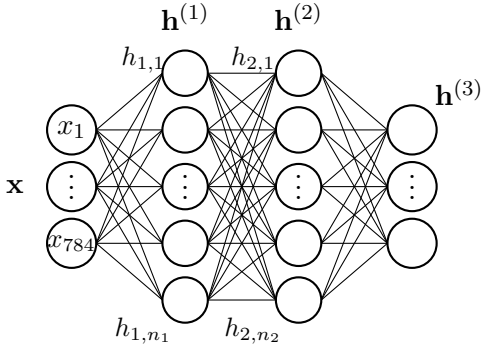


Figura 3.1: Red neuronal para modelar datos MNIST

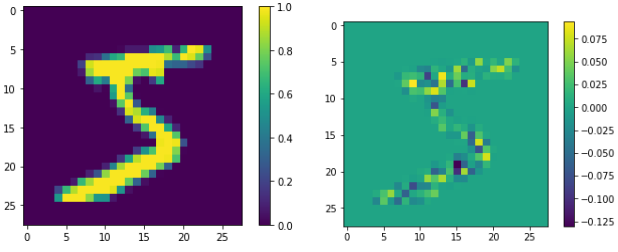
Observemos que en esta red establecemos conexiones entre cada variables de entrada: cada pixel de la imagen.

$$a_j^{(1)} = \sum_{i=1}^{m} x_i \cdot W_{ji} \tag{3.4}$$

Es decir, el modelo tiene embebido en sí la hipótesis de que cada pixel de la imagen está relacionado con cualquier otro en ella. Intuitivamente, esto debería de exponer algo sobre la falla en modelar imágenes con la formulación de la capa densa. Para hacer esto de forma explícita, se analiza una observación del conjunto de datos y las activaciones derivadas en el paso hacia delante de una red con esta arquitectura.

3.1.1. Entrenamiento y resultados del perceptrón en MNIST

Para esta sección se entrenó un perceptrón multicapas con la arquitectura propuesta en (3.2). Para dicho entrenamiento se escogió entropía cruzada (2.32) como función de pérdida, apropiada para problemas de clasificación multinomial. Durante el proceso de entrenamiento, el tamaño de los lotes fue quinientas observaciones para Gradiente Estocástico. Se entrenó por 50 épocas, es decir se muestreó del conjunto de entrenamiento el número de ejemplos su totalidad cincuenta veces. La probabilidad de dropout fue de 50 % por lo que cada observación del conjunto de entrenamiento fue utilizada, aproximadamente, 25 veces. Tras realizar este entrenamiento, el modelo logra tener una precisión de 95 % en el conjunto de validación. Las neuronas en la primera capa de este modelo computan una caracterización de la imagen basada en todos sus pixeles. Se analizan las activaciones de esta caracterización para la primera neurona de la capa, dado el ejemplo visualizado en la Figura 3.2.



(a) Un ejemplo del conjunto MNIST, dígito cinco.

(b) Activación del ejemplo $\mathbf{a}_0^{\prime(1)}$.

Figura 3.2: Ejemplo y activaciones de la primera neurona.

Recordemos la expresión para la primera preactivación de la red: $\mathbf{a}^{(1)} = \mathbf{W}^{(1)} \cdot [\mathbf{x}, 1]$. Ignorando la concatenación para el parámetro del sesgo se tienen las siguientes expresiones para la activación de la primera neurona,

$$\mathbf{a}^{(1)} = \mathbf{W}^{(1)}_{\cdot,:n_{l-1}} \cdot \mathbf{x} \implies \mathbf{a}^{(1)}_1 = \mathbf{W}^{(1)}_{1,:n_{l-1}} \cdot \mathbf{x} = \textstyle\sum_{j=1}^{n_{l-1}} W_{1,j} \cdot x_j. \tag{3.5}$$

Consideremos en vez el producto elemento a elemento para obtener un vector del tamaño de la imagen de entrada,

$$\mathbf{a}_0^{'(1)} = \mathbf{W}^{(1)}_{1,\cdot} \odot \mathbf{x} = \begin{bmatrix} W_{1,1} \cdot x_1 \\ \vdots \\ W_{1,n_{l-1}} \cdot x_{n_{l-1}} \end{bmatrix}. \tag{3.6}$$

Reacomodando $\mathbf{a}_0^{'(1)}$ como una matriz de 28×28, se puede visualizar esta matriz para analizar las preactivaciones en cada posición de los pixeles en la figura 3.2. Aun más, analizando el histograma de este vector observamos que sólo un 10 % de las activaciones son mayores a cero, i.e

$$\sum_{j=V} W_{0,j} \cdot x_j > 0, \tag{3.7}$$

con $V \subset [n_{l-1}]$. Después de este análisis, se puede pensar que esta forma de procesar datos con estructura de cuadrícula no es eficiente ni en cómputo ni en memoria[1]. Por lo descrito anteriormente, se busca reformular las preactivaciones que procesan imágenes buscando sólo relacionar pixeles en una vecindad en vez de usar la totalidad de pixeles en la imagen. Estas idea,

[1]El número de parámetros en un perceptrón multicapas para clasificación múltiple está dado por $m \times n + L - 1(n^2) + L \times n + n \times K + K$ considerando que las capas tienen el mismo número de neuronas. El término dominante de crecimiento es $L - 1(n^2)$ determinando el comportamiento exponencial. Esto indica ya la poca viabilidad de incrementar el número de neuronas o capas para analizar imágenes de mayor tamaño.

modelar la relación espacial de los pixeles, introduce la idea fundamental para una capa escondida distinta a la densa: la capa convolucional. A continuación se describe matemáticamente cómo la capa convolutiva hace uso de la hipótesis de correlación espacial.

3.2. Convoluciones

Dadas las motivaciones anteriores para reformular una capa escondida, buscamos incluir en el modelo la hipótesis de espacialidad, específicamente encontrar relaciones en localidades de la imagen. Consideremos una imagen entrada de tamaño m tal que $x \in \mathbb{R}^{n^{\frac{1}{2}} \times n^{\frac{1}{2}}}$. Definimos una localidad o vecindad de tamaño k como $\mathbf{v} = \mathbf{x}_{i:i+k,j:j+k}$. Buscamos extraer información de esta localidad de manera análoga a la formulación de la neurona presentada en (2.41). Para esto, consideremos la siguiente operación entre la vecindad $\mathbf{v}$ con un tensor de parámetros $\mathbf{k} \in \mathbb{R}^{k \times k}$,

$$\mathbf{v} * \mathbf{k} = \sum_p \sum_q v_{p,q} \cdot k_{p,q} \tag{3.8}$$

con $p, q \in \{1, ...k\}$. El tensor $\mathbf{k}$ usualmente es conocido como núcleo o *kernel*. La expresión anterior denota la operación sobre una sola vecindad, definamos esta operación sobre toda la imagen:

$$\mathbf{s}_{i,j} = (\mathbf{x} * \mathbf{k})_{i,j} = \sum_p \sum_q x_{i+p,j+q} \cdot k_{p,q} \tag{3.9}$$

Esta operación es llamada *correlación cruzada* y es similar a un producto punto entre matrices. Con esta operación descrita, se presenta la convolu-

ción discreta entre una matriz o tensor $\mathbf{x}$ y un kernel $\mathbf{k}$,

$$\mathbf{s}_{i,j} = (\mathbf{x} * \mathbf{k})_{i,j} = \sum_p \sum_q x_{i+p,j+q} \cdot k_{k-p,k-q} = \sum_p \sum_q x_{i+p,j+q} \cdot \tilde{k}_{p,q}. \quad (3.10)$$

El operador tilde, $\tilde{\mathbf{k}}$, corresponde a voltear las filas y las columnas de una matriz.

3.2.1. Capa Convolucional en Redes Neuronales

Recordando que se busca agregar información sobre una vecindad en una imagen, hacemos notar que las imágenes a color son representadas con tensores de dimensionalidad 3, $\mathbf{x} \in \mathbb{R}^{n^{\frac{1}{2}} \times n^{\frac{1}{2}} \times 3}$. La información de estos tensores en la tercera dimensión, profundidad o canales, representa el componente de algún color, rojo, verde o azul, de la imagen[2] En el caso de los datos MNIST las imágenes eran a blanco y negro, por lo tanto se tenía un solo canal: volúmenes de 28×28×1. Para agregar la información de los tres colores, el kernel descrito anteriormente será entonces también un tensor con tres dimensiones, $\mathbf{k} \in \mathbb{R}^{\mathrm{k} \times \mathrm{k} \times 3}$. Considerando las nuevas dimensiones de las imágenes y algún kernel, redefinimos la convolución discreta para imágenes,

$$\mathbf{s}_{i,j} = (\mathbf{x} * \mathbf{k})_{i,j} = \sum_d \sum_p \sum_q x_{i+p,j+q,d} \cdot k_{k-p,k-q,d}, \quad (3.11)$$

con $d \in \{1, 2, 3\}$.

La dimensionalidad del resultado de una convolución es $\mathbf{x} * \mathbf{k} \in \mathbb{R}^{W \times H \times 1}$. Más adelante se detalla el cálculo de W y H, por ahora notemos que la

[2] A este formato se le llama RGB por sus siglas en inglés Red, Green, Blue.

profundidad del resultado es unitaria. El tensor resultante de (3.11) es una caracterización del tensor de entrada dado *un* kernel $\mathbf{k}$ en particular y se comúnmente se le llama filtro[3]. En aprendizaje profundo se busca aprender varias caracterizaciones del mismo tensor a través de distintos filtros. El propósito de esto es que cada filtro aprenda a detectar bordes en distintos ángulos o manchas, estos filtros son aprendidos a través de Gradiente Estocástico. Así, la capa convolucional usada en una red neuronal para procesamiento de imágenes resulta en un tensor de dimensionalidad tres definido por la siguiente expresión,

$$\text{Conv}(\mathbf{x}, \{\mathbf{k}_i\}_{i=0}^{D-1}) = [\mathbf{x} * \mathbf{k}_1, \ldots, \mathbf{x} * \mathbf{k}_{D-1}] \in \mathbb{R}^{W \times H \times D}, \tag{3.12}$$

donde D es el número de filtros propuestos en la capa. Una forma de visualizar la capa convolutiva es pensar en el kernel $\mathbf{k}$ como un cubo que agrega información sobre la imagen, en otro tensor, mientras la recorre a lo largo y ancho. La Figura 3.3 es ilustrativa de este proceso.

[3]Esta caracterización o filtro es conocida como *feature map* en inglés.

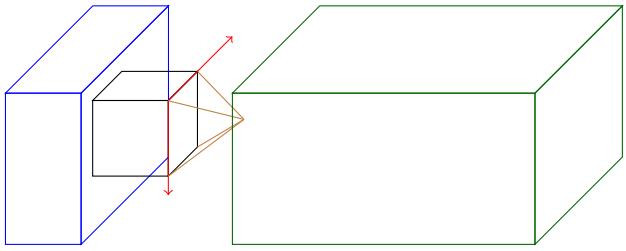


Figura 3.3: Visualización de capa convolutiva: datos de entrada en azul (imagen), en negro el kernel, las flechas rojas indican el movimiento del kernel a lo largo y alto de la imagen. Por último, en verde, el tensor resultante de la capa para varios filtros.

El resultado de una capa convolucional como la descrita en (3.12) depende de dos detalles de implementación:

- Las posiciones que el kernel recorre con cada movimiento al recorrer el tensor de entrada a lo alto y ancho durante la operación. A esta cantidad se le denomina paso o *stride*.
- El número de elementos que se agregan al borde del tensor de entrada, a esta técnica se le conoce como *padding*. Esta técnica permite considerar los elementos de los bordes de los tensores de entrada pero ayuda a preservar el tamaño de este.

Ahora bien, consideremos el tensor de entrada $\mathbf{x} \in \mathbb{R}^{n^{\frac{1}{2}} \times n^{\frac{1}{2}}}$ y un *kernel* $\mathbf{k} \in \mathbb{R}^{k \times k}$. Declarando W y H como el ancho y alto de tensores, entonces tenemos que $H = W$. Entonces se tienen las siguientes expresiones para las

dimensiones del tensor resultante de una convolución,

$$W_c = \frac{W-k+2P}{S} + 1 \quad H_c = \frac{H-k+2P}{S} + 1 \tag{3.13}$$

con P el tamaño del *padding* y S el paso o *stride*.

En una red convolucional típica, la arquitectura de esta consta de capas **Conv** seguidas otros de dos tipos de capas: capa de activación, como descrita en 2.5, y enseguida una capa de agregación o *pooling*. Este último tipo de capa operan recorriendo el tensor de entrada a lo largo y a lo ancho, análogamente a las capas convolucionales, aplicando a vecindades funciones de agregación como el máximo, la media, la norma L^2, entre otras. El objetivo de estas capas es proveer al modelo de la propiedad de invarianza a traslaciones [Goodfellow et al., 2016] así como reducir el tamaño del argumento de entrada, reduciendo así la dimensionalidad del modelo. Estas capas actúan independientemente para cada entrada de la dimensión de profundidad del tensor de entrada. Una configuración común para este tipo de capa consta de un filtro de tamaño 2x2 y un paso de 2 para no superposicionar los elementos de una operación a otra con la operación máximo.

Observemos que la capa convolucional ofrece una mejora computacional y en memoria: el costo de computar (3.10) tiene un orden de $\mathcal{O}(k \times n)$ y requiere almacenar k^2 parámetros. El costo de calcular una capa densa, (2.41), es $\mathcal{O}(n^2)$ y almacena n^2 parámetros. Considerando que k usualmente es un número entre 2 y 10 a comparación de n que podría estar entre 250 y 1000, la capa convolucional de hecho presenta una mejora en cómputo y almacenamiento. El cambio en la cantidad de parámetros de una capa

densa a la convolutiva se da gracias al hecho de que en la capa convolutiva, los parámetros se comparten para cada posición del tensor de entrada. Al compartir el tensor de parámetros para todo el tensor de entrada estamos explícitamente modelando que las relaciones se dan en localidades.

En este capítulo describimos una arquitectura particularmente adecuada para el procesamiento de imágenes dado que su estructura refleja relaciones espaciales. Esta arquitectura es útil para modelar las imágenes relacionadas a capturas de pantalla. Sin embargo, esta no es adecuada para el modelaje del código que las genera ya que no poseen dicha estructura. Para estos datos se necesita una arquitectura que explote la naturaleza secuencial de los datos de texto. A continuación se presenta una arquitectura que asume estructura temporal en ellos para procesarlos de manera adecuada y eficiente.

3.3. Modelado de secuencias con redes recurrentes

Notación para el resto del trabajo. A partir del siguiente capítulo los superíndices entre paréntesis en escalares o vectores denotan la dimensión asociada a la temporalidad y no potencias. E.g. $y^{(1)}$ denota el valor de la variable y al tiempo 1.

La naturaleza secuencial del código que genera pantallas demanda que se use arquitecturas de redes neuronales distintas a la convolucional dada su estructura fundamentalmente distinta a una cuadrícula. El código puede ser representado como una serie de elementos "en el tiempo". Tomando esto en

consideración definamos un conjunto de datos con dependencia secuencial:

$$\begin{aligned} \mathcal{D} &= \{\xi_i^{(t)} := (\mathbf{x}_i^{(t)}, y_i^{(t)})\}_{i=0}^{m} \in \mathbb{R}^n \times [K] : t \in [T], \\ y_i^{(t)} &= f(\mathbf{x}_i^{(t)}, \{\xi_i^{(t')}\}_{t'=0}^{t-1}). \end{aligned} \tag{3.14}$$

En el caso del conjunto de datos de código para generar interfaces, t no denota temporalidad, en vez define la posición de un elemento en la secuencia de *tokens*[4]. Las redes recurrentes (RNNs) son modelos neuronales que consideran, de forma explícita en su formulación, hipótesis de temporalidad en los datos. Estas pueden manejar secuencias de tamaño variable permitiéndoles escalar para distintos problemas así como capturar patrones dependientes de la temporalidad, a diferencias de otras formulaciones de redes neuronales más estáticas. A continuación se presenta una formulación de una red recurrente base para luego extenderla y construir la formulación de la red LSTM [Hochreiter and Schmidhuber, 1997]. Se explora este modelo y su construcción matemática para contextualizar el trabajo de [Beltramelli, 2017] donde se hace uso de él para procesar de las secuencias de código.

3.3.1. Formulación y representación gráfica

Los modelos recurrentes, como los convolucionales, hacen uso de parámetros compartidos, sin embargo estos lo hacen para cada valor de t. A modo de ilustrar la recurrencia en una función, proponemos un posible modelaje de $y_i^{(t)}$. Sea $\xi_i^{(t)} = (\mathbf{x}_i^{(t)}, y_i^{(t)}) \in \mathcal{D}$, entonces se propone un modelo que con-

[4]Un token puede expresar un elemento de alguna secuencia de código. Este puede ser una variable, parte de una instrucción como “double”, “row”, “{” , “var1”

sidere la dependencia temporal entre observaciones de la siguiente manera,

$$\mathbf{s}^{(t)} = g(\mathbf{s}^{(t-1)}, \xi_i^{(t)}), \quad \hat{y}_i^{(t)} = h(\mathbf{s}^{(t)}). \tag{3.15}$$

La dependencia temporal está modelada por la definición de $\mathbf{s}^{(t)}$. Se puede representar el modelo en (3.15) con el grafo en la Figura 3.4.

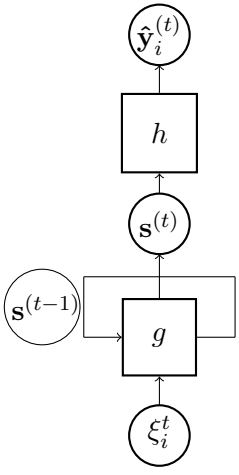


Figura 3.4: Grafo recurrente asociado a 3.15

El ciclo en el grafo es la forma de representar la recurrencia del modelo, la dependencia temporal entre observaciones actuales $\xi_i^{(t)} \in \mathcal{D}$ y una representación de pasadas $\mathbf{s}^{(t)}$. A manera de ejemplo, para $t = 2$ tenemos la siguiente descomposición, $\hat{\mathbf{y}}^{(2)} = h(\mathbf{s}^{(2)}) = h(g(\mathbf{s}^{(1)}, \xi_i^{(1)})) = h(g(g(\mathbf{s}^{(0)}, \xi_i^{(0)}), \xi_i^{(1)}))$. Al inicializar $\mathbf{s}^{(0)} = 0$, por ejemplo, se tiene una expresión definida y, dada esta recurrencia, el modelo procesa información sobre observaciones pasadas. En el grafo, esta descomposición equivale a desdoblar el ciclo en t pasos, podemos ver el grafo desdoblado en la Figura

3.5.

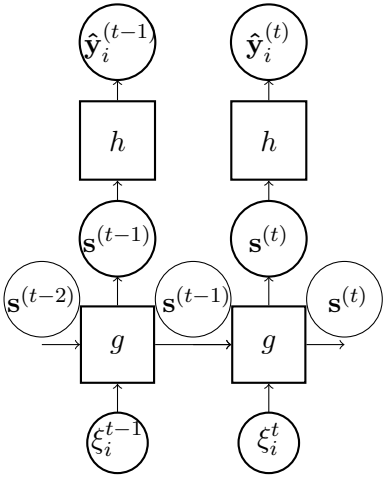


Figura 3.5: Grafo de red recurrente desdoblada para $t-1$ y t

Un ejemplo de red recurrente.

Basándonos en la forma de modelar recurrencia presentada en (3.15), se muestra a continuación un ejemplo de una red recurrente, un modelo de clasificación, modelando $y_i^{(t)} \sim h(\mathbf{x}_i^{(t)}|\mathbf{\Theta})$ para el conjunto $\mathcal{D}$ definido en (3.14). El modelo deberá proponer una clase para y_i para cada paso en el tiempo, es decir para cada valor de $t \in [T]$. Las siguientes ecuaciones definen el modelo y la forma de computar el paso hacia delante para cada valor de t,

$$\begin{aligned} \mathbf{a}^{(t)} &= \mathbf{b} + \mathbf{W}\mathbf{s}^{(t-1)} + \mathbf{V}\mathbf{x}^{(t)}, & \mathbf{s}^{(t)} &= g(\mathbf{a}^{(t)}) \\ \mathbf{o}^{(t)} &= \mathbf{c} + \mathbf{U}\mathbf{s}^{(t)}, & \hat{y}^{(t)} &= \text{softmax}(\mathbf{o}^{(t)}). \end{aligned} \tag{3.16}$$

Para este modelo, el conjunto de parámetros está definido por $\mathbf{\Theta} = \{\mathbf{W}, \mathbf{V}, \mathbf{U}, \mathbf{b}, \mathbf{c}\}$.

Analicemos las ecuaciones del modelo para entender su forma y funcionalidad,

- $\mathbf{a}^{(t)}$ es la preactivación, notemos que depende de $\mathbf{s}^{(t-1)}$ y de $\mathbf{x}^{(t)}$, esta es la forma en que el modelo considera la dependencia temporal de los datos. Los parámetros $\mathbf{W}$ y $\mathbf{V}$ tienen el propósito de incluir, retener o descartar información de observaciones anteriores así como de la actual.

- $\mathbf{s}^{(t)}$ es comúnmente conocido como el estado oculto del sistema. Este es la representación de toda la información del pasado y, dependiendo de la función de pérdida, decidirá qué retener y qué olvidar.

- $\mathbf{W}$ es la conexión, a través de t, de un estados oculto a otro y $\mathbf{V}$ la conexión de la observación actual a la preactivación, y consecuentemente al estado oculto, al tiempo t.

- $\mathbf{b}$ y $\mathbf{c}$ son parámetros de sesgo del modelo.

- $\mathbf{o}^{(t)}$ es la salida de la red para cada paso en el tiempo. $\mathbf{U}$ representa la conexión entre el estado oculto y la predicción.

- Para el caso de clasificación, se usa $\mathbf{o}_t$ como argumento de una regresión *softmax* cuyo resultado será la predicción $\hat{y}^{(t)}$.

En teoría el modelo presentado en (3.16) podría aprender relaciones temporales de larga dependencia, i.e. $\xi_i^{(t)} \sim \xi_i^{(t-q)}$ y $t > t - q$. En la práctica se encontró [Hochreiter and Schmidhuber, 1997] que tanto arquitecturas como esta así como los métodos para entrenarlas basados en gradiente tienen dificultades para encontrar modelos que logren replicar dependencias

temporales de largo plazo. La raíz de la problemática anterior tiene dos razones principales:

1. El problema de gradiente desvaneciente/explotante es particularmente difícil de resolver en RNNs dada la profundidad de la red resultante al ser desdoblada.

2. Durante entrenamiento, los parámetros del modelo reciben actualizaciones conflictivas entre sí al tratar de aprender a retener y, para otros pasos, descartar información. En el caso anterior, $\mathbf{W}$ tendría problemas de estabilización.

Estos obstáculos son la motivación para la el modelo LSTM[Hochreiter and Schmidhuber, 1997]. Esta arquitectura de red recurrente resuelve ambas problemáticas con elegantes recursos matemáticas.

3.3.2. Aprendizaje de dependencias de largo plazo con el modelo LSTM

El modelo propuesto en [Hochreiter and Schmidhuber, 1997] ataca las problemáticas descritas al final de la Sección 3.3.1, siendo capaz de aprender dependencias de largo plazo. Para resolver el primer obstáculo, el problema de gradiente desvaneciente/explotante, los autores hacen notar que para mantener el gradiente constante propagándose hacia atrás durante entrenamiento, se tiene que cumplir:

$$g'(\mathbf{x} \cdot \mathbf{w})w_j = 1 \tag{3.17}$$

para algún parámetro w_j y g la función de activación. Al resolver esta ecuación diferencial se obtiene que

$$g(\mathbf{x} \cdot \mathbf{w}) = \frac{\mathbf{x} \cdot \mathbf{w}}{w_j} \tag{3.18}$$

lo cual indica que g tiene que ser una función lineal y que la activación tiene que ser constante. Esta idea fundamental para el modelo propuesto los autores la denominan *Carrusel de Error Constante*, CEC. El segundo elemento novedoso en el modelo LSTM resuelve el problema de conflictos durante la actualización de parámetros, la compuerta de información. Estas estructuras matemáticas permiten regular la salida/entrada de información adicional a un flujo, distinto, de información. En los modelos recurrentes, el estado oculto es el flujo que debe admitir o rechazar nueva información en el momento que lo necesite, de un valor de t a otro por ejemplo. Una compuerta de información es, de hecho, una versión localizada de una red neuronal con función de activación sigmoide,

$$\mathbf{c}^{(t)} = \sigma(\mathbf{x}^{(t)} \cdot \mathbf{w}), \quad \mathbf{s}^{(t)} = \mathbf{s}^{(t-1)} \odot \mathbf{c}^{(t)}. \tag{3.19}$$

Al usar el producto elemento a elemento y dado que la imagen de la función sigmoide es $[0, 1]$ se logra la regulación de información de, en este caso, un estado oculto al siguiente. Con estos dos elementos y sus funcionalidades definidas, a continuación se presenta la definición matemática y propiedades de la célula LSTM.

Definición de célula LSTM

Usando compuertas de información para retener, descartar o proponer nueva información el modelo LSTM consta de cuatro de ellas para dar lugar a

una célula de memoria:

$$\begin{aligned}
\mathbf{f}^{(t)} &= \sigma(\mathbf{W}_f \cdot [\mathbf{s}^{(t-1)}, \mathbf{x}^{(t)}] + \mathbf{b}_f), & \mathbf{i}^{(t)} &= \sigma(\mathbf{W}_i \cdot [\mathbf{s}^{(t-1)}, \mathbf{x}^{(t)}] + \mathbf{b}_i), \\
\tilde{\mathbf{C}}^{(t)} &= \tanh(\mathbf{W}_C \cdot [\mathbf{s}^{(t-1)}, \mathbf{x}^{(t)}] + \mathbf{b}_C), & \mathbf{C}^{(t)} &= \mathbf{f}^{(t)} \odot \mathbf{C}^{(t-1)} + \mathbf{i}^{(t)} \odot \tilde{\mathbf{C}}^{(t)}, \\
\mathbf{o}^{(t)} &= \sigma(\mathbf{W}_o \cdot [\mathbf{s}^{(t-1)}, \mathbf{x}^{(t)}] + \mathbf{b}_o), & \mathbf{s}^{(t)} &= \mathbf{o}^{(t)} \odot \tanh(\mathbf{C}_t),
\end{aligned} \tag{3.20}$$

por lo que el conjunto de parámetros del modelo está dado por $\mathbf{\Theta} = \{\mathbf{W}_f, \mathbf{W}_i, \mathbf{W}_C, \mathbf{W}_o, \mathbf{b}_f, \mathbf{b}_i, \mathbf{b}_C, \mathbf{b}_o\}$. Bajo esta formulación, $\mathbf{o}_t$ es la salida de la célula y $\mathbf{f}^{(t)}, \mathbf{i}^{(t)}, \tilde{\mathbf{C}}^{(t)}$ son compuertas de información que interactúan con el estado oculto $\mathbf{s}^{(t)}$ para actualizarlo de distintas maneras,

- $\mathbf{f}^{(t)}$ es llamada compuerta de olvido. Esta compuerta decide cuánta información dejar de retener en el estado considerando el estado anterior y $\mathbf{x}^{(t)}$
- $\mathbf{i}^{(t)}$ es llamada compuerta de entrada. Esta compuerta decide cuánta información actualizar al estado dado el estado anterior y el elemento actual de la secuencia
- $\tilde{\mathbf{C}}^{(t)}$ es llamada compuerta de propuesta. Esta compuerta propone candidatos nuevos para agregar al estado.

La ecuación (3.20) define una sola célula LSTM. Usualmente, las arquitecturas que hacen uso de este tipo de capas implementan más de una célula, en uno o varios bloques. Estos bloques pueden incluso apilarse uno sobre el otro para usar la salida de uno como argumento del otro dando lugar a una jerarquía de capas y crear así representaciones más complejas.

3.4. Entrenamiento de redes recurrentes

Recordemos que para llevar acabo el entrenamiento de una red neuronal usando métodos basados en el gradiente se necesita determinar la función de pérdida o costo del modelo. En el caso de una red recurrente la formulación de esta debe incluir la estructura temporal del problema. El costo para estas redes es la suma de los costos individuales por cada observación por cada paso en el tiempo,

$$J(\theta) = \sum_{i}^{m} \sum_{t}^{T} l(h(\mathbf{x}_i^{(t)}|\theta), \hat{y}_i^{(t)}) \tag{3.21}$$

con l la función de pérdida para clasificación o regresión.

El hecho de desdoblar una red recurrente permite a este tipo de modelos manejar secuencias de longitud variable. Sin embargo esto requiere técnicas especiales en la practica. Durante entrenamiento, usando gradiente estocástico, se proporcionan lotes de secuencias cuyas longitudes deben coincidir para poder desdoblar la red el mismo número de veces por observación. Así, debe definirse la longitud de las secuencias que serán vistas durante entrenamiento y, si es necesario, usar técnicas de padding para hacer estas coincidir.

3.4.1. Gradiente de redes recurrentes

Como se mencionó en la Sección 3.3, una red recurrente puede replantearse como el desdoblamiento de esta a través del tiempo, una red con T capas, una para cada paso de la secuencia. Esto no sólo ayuda a una alternativa de formulación también provee de un marco de referencia para

entrenarlas. Con una red con T número de capas fijas dado el desdoblamiento de su formulación recurrente se puede usar propagación hacia atrás para encontrar el gradiente de sus parámetro y actualizarlos con Gradiente Estocástico. A este método de entrenamiento se conoce como propagación hacia atrás en el tiempo o abreviado *BPTT* por sus siglas en inglés backpropagation through time. En este sentido, se puede pensar en las RNNs como redes no recurrentes de gran profundidad que comparten parámetros para sus distintas capas en la dimensión del tiempo. Sin embargo, el cálculo para la actualización de un sólo parámetro sigue siendo costoso computacionalmente por lo que en la práctica se usa una versión de BPTT llamada *BPTT truncado*.

BPTT truncado computa el estado oculto y la salida de la red para cada valor de t, el paso hacia delante. Cada l_1 iteraciones del paso hacia delante se lleva a cabo BPTT con l_2 pasos hacia atrás en el tiempo. Esto hace que la actualización de parámetros sea computacionalmente barata si l_2 es pequeño al mismo tiempo permitiendo al estado oculto "ver" información considerablemente atrás en el tiempo que puede ser utilizada a manera que se necesite. BPTT truncado se detalla en el Algoritmo 2. Con los mo-

Algoritmo 2: BPTT truncado

```
for t ∈ {1, ..., T} do
    calcular s^(t), o^(t);
    if t ≡ 0 (mód l_1) then
        efectuar BPTT para t → t − l_2
```

delos expuestos anteriormente, hipótesis embebidas sobre la estructura de

los datos a procesar así como formas de entrenarlos, se cuenta con los elementos necesarios para describir el modelo que aproxima una solución para el problema de generación de código dada una interfaz gráfica, el modelo pix2code.

Capítulo 4

Pix2Code

Este capítulo se enfoca en el estudio del trabajo de [Beltramelli, 2017], programación inducida a partir de una captura de pantalla. Se analiza el problema planteado, los datos con los que se cuenta, el modelo propuesto para aproximar una solución, formas de entrenamiento y resultados obtenidos tanto en el trabajo original como en la replicación llevada a cabo para este trabajo. El problema planteado en el trabajo original se puede formular con la siguiente pregunta. Dada la imagen de una interfaz gráfica, ¿Se puede obtener el código que genera dicha interfaz usando un modelo de aprendizaje profundo? Para resolver dicho problema se cuentan con los datos usados por el autor, mismos que se publicaron a la par del trabajo original. Para dar el contexto adecuado, se describe el conjunto de datos y la propuesta de modelo original del autor. La arquitectura de dicho modelo tiene como elementos centrales la capa convolucional y la capa LSTM para modelar datos de imágenes y el código asociado a ellas, respectivamente.

Este modelo es denominado por el autor *pix2code*. Así mismo, se presentan particularidades del entrenamiento llevado a cabo en el trabajo de [Beltramelli, 2017], así como los resultados obtenidos en la replicación de este y la comparación a modelos inspirados en el trabajo original pero de distinta arquitectura.

4.1. Descripción del problema y datos

El problema de condicionar un modelo a la observación de una interfaz gráfica para generar el código asociado a ella es un problema similar al problema de descripción de imágenes. Los trabajos de [Karpathy and Li, 2014] y [Xu et al., 2015] muestran que la solución a este problema, a través de modelos de aprendizaje profundo, es factible. En palabras, se cuenta con una imagen que se busca describir dado cierto marco de referencia ya sea un lenguaje escrito o computacional. Para el caso en cuestión, dada una imagen y una secuencia de tokens previos se busca predecir el token siguiente. Realizando esta predicción de forma iterativa se genera la totalidad del código que da lugar a dicha interfaz.

De forma matemática, se busca resolver un problema de clasificación. Declarando $\mathbf{x}$ la imagen de una interfaz y $\mathbf{y} = [y^{(1)}, \ldots, y^{(T)}]$ el código asociado a la interfaz, una secuencia de tokens de longitud T, entonces buscamos predecir la clase de $y^{(t+1)}$ condicionado a $\mathbf{x}$ y $[y^{(1)}, \ldots, y^{(t)}]$, para todo $t \in [T]$. Es decir, se busca un modelo que estime

$$P(y^{(t+1)} = k | [\mathbf{x}, [y^{(1)}, \ldots, y^{(t)}]]), \tag{4.1}$$

con $k \in [K]$ los distintos tokens posibles. Para generar un código compilable

se necesita contar con una estructura sintáctica correcta. Para lograr esto, es necesario contar con información sobre el código hasta el paso t de forma que se mantenga dicha estructura. Esta es la razón por la que la predicción del token siguiente está condicionada a los anteriores.

Con esta descripción del problema y la forma de modelarlo se continua describiendo el conjunto de datos, su generación y la declaración matemática de este.

La descripción a continuación corresponde al conjunto de datos publicado por [Beltramelli, 2017]. Las secuencias de código y consecuentemente las imágenes de interfaces gráficas se generaron de forma sintética a través de programación. Este proceso da lugar a observaciones que constan de parejas interfaz gráfica - código, las cuales están claramente asociadas. Este proceso se llevó a cabo para tres plataformas móviles, web, iOS y Android, dando lugar a tres conjuntos de datos. Aunque el autor publicó tres conjuntos, en este trabajo solo se consideró el conjunto de interfaces web. Es importante aclarar también que dichas secuencias de código son en realidad un mapeo del código nativo de estas plataformas a un *lenguaje de dominio específico* o LDE. En general, los LDEs son más restrictivos que los lenguajes nativos. Esto logra dos cosas: (1) reducen la complejidad del lenguaje original y (2) reducen el espacio de búsqueda de tokens. Es también relevante notar que la predicción del modelo no es a nivel caracter, los tokens a predecir son pequeños bloques de código. Se puede encontrar un ejemplo de las secuencias de código y de capturas de pantalla en las Figuras 4.3 y 4.4 respectivamente.

Se recuerda que $\mathbf{x}_i$ representa una imagen de una interfaz gráfica i

y $\mathbf{y}_i$ la secuencia de tokens de longitud T que genera esta. Representar las secuencias de código como un vector de tokens, $\mathbf{y}_i$, es conveniente para una conceptualización teórica. Sin embargo, durante entrenamiento del modelo se usan vectores mapeo token-clase. Es decir, se mapea el espacio de secuencias de tokens a un espacio ralo de vectores cuyas entradas son identificadores. Estos identificadores son números enteros correspondiente a tokens en la secuencia original. De esta forma asignando clases únicas a cada token distinto. Considerando esto, la representación matemática del conjunto de datos usados para entrenamiento e inferencia es descrito por la siguiente ecuación,

$$\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=0}^{m} \subset \mathbb{R}^{n^{\frac{1}{2}} \times n^{\frac{1}{2}}} \times [K]^T, \tag{4.2}$$

con K el número de tokens distintos en $\{\mathbf{y}_i\}_{i=0}^{m} \subset \mathcal{D}$. Este mapeo entre los distintos tokens y $[K]$ se conoce como diccionario ya que contiene cualquier token posible en las secuencias de código y la clase asociada a este. Para el conjunto de datos web, se cuenta con un total de diecinueve clases distintas a predecir. Se cuenta con dos conjuntos de datos, uno de entrenamiento y otro de prueba. El tamaño de estos se puede consultar en la Tabla 4.1. Con la descripción del problema y los datos que se cuentan para resolverlo, se continua este capítulo elaborando sobre el modelo propuesto en el trabajo original.

Conjunto	Observaciones
entrenamiento	85756
prueba	14265

Cuadro 4.1: Tamaño de conjuntos de entrenamiento y prueba

4.2. Arquitectura Pix2code

En [Beltramelli, 2017] se usan los datos descritos en la sección 4.1 para resolver el problema descrito en la misma y formulado en (4.1), proponiendo el modelo denominado pix2code. Este consta de una arquitectura que procesa la imagen y el código a través de dos submodelos independientes. Considerando una tupla $(\mathbf{x}, \mathbf{y}) \in \mathcal{D}$ (definido en (4.2)), entonces un modelo convolucional procesa $\mathbf{x}$ y un modelo recurrente $\mathbf{y}$, creando una representación de ambas entradas para ser usadas en conjunto. Una segunda capa LSTM toma ambas representaciones y actúa como decodificador cuya salida es un vector denso. Este vector será el argumento de una regresión softmax, la salida de esta es la predicción del modelo. Para una visualización de este se hace referencia a la Figura 4.1.

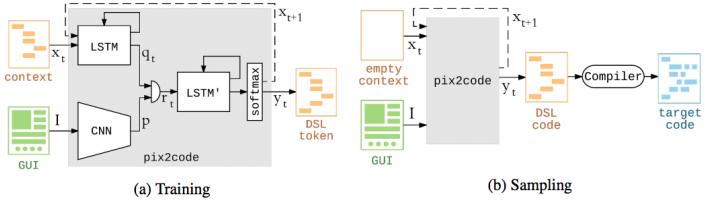


Figura 4.1: Vista general del modelo pix2code. Fuente de imagen [Beltramelli, 2017]

El modelo visual tiene una arquitectura que usa capas convolucionales,

Pooling con operación máximo y Dropout:

$$\begin{aligned}
&\mathbf{c}_1 = \text{RELU}(\text{Conv}(\mathbf{x}, \{\mathbf{k}_{1i}\}_{i=1}^{D_1})), && \mathbf{c}_2 = \text{RELU}(\text{Conv}(\mathbf{c}_1, \{\mathbf{k}_{2i}\}_{i=1}^{D_1})),\\
&\mathbf{m}_1 = \text{Maxpool}^{\dagger}(\mathbf{c}_2, (2,2)), && \mathbf{c}_3 = \text{RELU}(\text{Conv}(\mathbf{m}_1, \{\mathbf{k}_{3i}\}_{i=1}^{D_2})),\\
&\mathbf{c}_4 = \text{RELU}(\text{Conv}(\mathbf{c}_3, \{\mathbf{k}_{4i}\}_{i=1}^{D_2})), && \mathbf{m}_2 = \text{Maxpool}^{\dagger}(\mathbf{c}_4, (2,2)),\\
&\mathbf{c}_5 = \text{RELU}(\text{Conv}(\mathbf{m}_2, \{\mathbf{k}_{5i}\}_{i=1}^{D_3})), && \mathbf{c}_6 = \text{RELU}(\text{Conv}(\mathbf{c}_5, \{\mathbf{k}_{6i}\}_{i=1}^{D_3})),\\
&\mathbf{m}_3 = \text{Maxpool}^{\dagger}(\mathbf{c}_6, (2,2)), && \mathbf{d}_1 = \text{RELU}^{\dagger}(\mathbf{W}_1 \cdot [\mathbf{m}_3, 1]),\\
&\mathbf{MV} = \text{RELU}^{\dagger}(\mathbf{W}_2 \cdot [\mathbf{d}_1, 1]).
\end{aligned} \tag{4.3}$$

Similarmente a la Sección 3.1, el superíndice † indica una operación Dropout sobre la salida de la operación en cuestión. Los parámetros del modelo visual tienen las siguientes especificaciones, $k_{1i}, k_{2i}, k_{3i}, k_{4i}, k_{5i}, k_{6i} \in \mathbb{R}^{3\times 3}$, $D_1 = 32$, $D_2 = 64$, $D_3 = 128$ y $\mathbf{W}_1, \mathbf{W}_2$ son los parámetros de una capa densa con 1024 neuronas cada una. El resultado de este submodelo, un vector de longitud 1024, denominado **MV** (modelo visual) será usado en conjunto con la salida del modelo de lenguaje por un decodificador recurrente.

El modelo de lenguaje se encarga de aprender representaciones vectoriales del código para cada valor de t. Este tratará de capturar dependencias temporales que encuentre en las secuencias de tokens. El submodelo consta de dos capas LSTM apiladas una sobre otra,

$$\mathbf{l} = \text{LSTM}(\mathbf{y}, 128), \quad \mathbf{LM} = \text{LSTM}(\mathbf{l}, 128). \tag{4.4}$$

El modelo de lenguaje procesa las secuencias de tokens, creando una representación de este en un vector de tamaño 128.

La salida de los modelos de visión y de lenguaje es concatenada y proporcionada como argumento de entrada a otra capa LSTM que a su vez

tiene otra capa como esta encima de ella. A este submodelo se le denomina decodificador,

$$\mathbf{conj} = [\mathbf{MV}, \mathbf{LM}], \quad \mathbf{dc} = \text{LSTM}(\mathbf{conj}, 512), \quad \mathbf{dec} = \text{LSTM}(\mathbf{dc}, 512). \tag{4.5}$$

La salida del decodificador, **dec**, será usada para llevar a cabo la regresión softmax que propone el token siguiente dada la imagen y la secuencia. Así, la predicción del modelo pix2code puede ser representado de la siguiente manera,

$$\hat{y}^{(T+1)} = \text{pix2code}(\mathbf{x}, \mathbf{y}) = \arg\max_{k \in [K]} \{\text{softmax}(\mathbf{dec}([\mathbf{MV}(\mathbf{x}), \mathbf{LM}(\mathbf{y})]))\}. \tag{4.6}$$

Con el modelo definido, se necesita entrenarlo usando los datos, medir el desempeño de este y hacer inferencia sobre nuevos datos.

4.3. Entrenamiento, inferencia y resultados

En el trabajo de [Beltramelli, 2017] se usó un método de optimización basado en Gradiente Estocástico llamado propagación de raíz cuadrada media[1] cuyo propósito es también resolver un problema de minimización haciendo uso del gradiente de una función objetivo de forma similar a GE. Así, se evalúa la entropía cruzada (2.32) entre las predicciones y las clases reales para usar como función de pérdida. Durante el proceso de entrenamiento el modelo procesó imágenes de tamaño 256 y secuencias de código

[1]Este método es usualmente conocido como RMSProp por sus siglas en inglés, Root Mean Square Propagation

con longitud 48, i.e. $\mathbf{x}_i \in \mathbb{R}^{256\times256}$ y $\mathbf{y}_i \in [K]^{48}$. La longitud de las secuencias se determinó por el autor tras realizar experimentos que probaron que dicha longitud era un buen punto de balance entre cómputo y distancia necesaria para capturar dependencias temporales de largo plazo.

Adicionalmente, las secuencias de código fueron modificadas para incluir tokens especiales 'INICIO', 'FIN' al inicio y al final de las secuencias respectivamente. En [Karpathy and Li, 2014], se utilizó y probó útil esta técnica para ayudar al modelo reconocer el inicio y fin de una secuencia. Durante este entrenamiento, se usaron mini-lotes de 64 observaciones. En el trabajo original, se entrenó el modelo por 10 épocas durante las cuales se observó la función de pérdida. Como parte de la experimentación alternativa propuesta en el trabajo de esta tesis, se extendió el entrenamiento hasta 25 épocas. También, se evaluó en cada época la distancia de Levenshtein. Esta distancia puede formularse como el mínimo número de cambios (substitución, supresión o inserción) entre dos cadenas de caracteres. En la Figura 4.2 podemos observar la evolución de éstas dos métricas así como la precisión de clasificación través de las épocas.

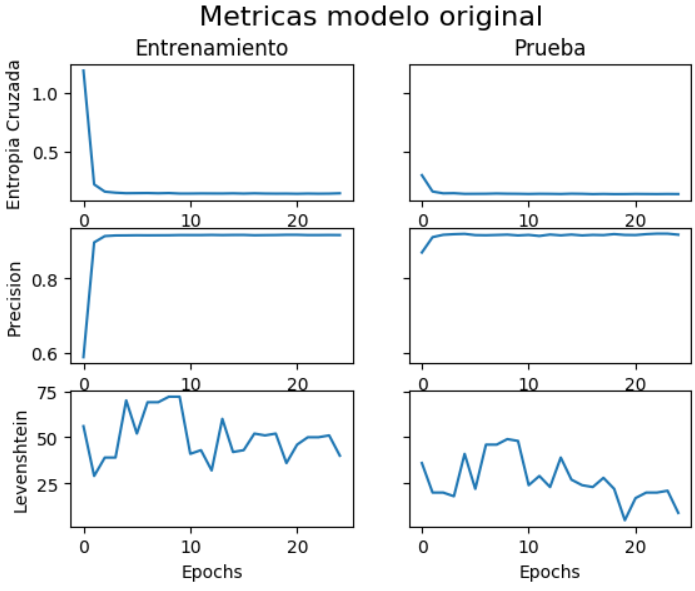


Figura 4.2: Métricas a través de las épocas modelo original

4.4. Inferencia y ejemplos

Para efectuar inferencia sobre una imagen de interfaz gráfica, el modelo se alimenta con dicha imagen y una secuencia inicial de código. Esta secuencia inicial es de longitud 48 y consta de tokens inicializados vacíos a excepción del token "INICIO". Una vez que se cuenta con la predicción, se agrega a la secuencia de entrada y se vuelve a efectuar inferencia sobre la imagen y la nueva secuencia de código. El proceso anterior se repite hasta predecir el token "FIN" indicando la totalidad de la secuencia. Una vez alcanzado este escenario, el proceso de inferencia para la interfaz gráfica termina. Por último, para generar la interfaz gráfica asociada al código in-

ferido se utilizan técnicas de compilación provistas en [Beltramelli, 2017]. Se visualiza un ejemplo de inferencia de código generado por el modelo en la Figura 4.3 y las interfaces gráficas correspondiente en la Figura 4.4. Es importante notar que el texto asociado a botones o etiquetas es generado de forma aleatoria y por lo tanto no es involucrado en la función de pérdida o métricas de evaluación del modelo. Dado que este modelo muestra la capacidad de aprender a generar el código asociado a una interfaz gráfica dada, es natural pensar en distintas arquitecturas y estudiar su desempeño. En la siguiente sección, se proponen dos arquitecturas alternativas a entrar con el mismo conjunto de datos y se comparan con los resultados de la arquitectura original.

4.5. Arquitecturas alternativas para pix2code

Dados los resultados al replicar el entrenamiento del modelo de [Beltramelli, 2017], se proponen dos arquitecturas distintas y se presentan los resultados obtenidos, comparándolos con los presentados en la sección anterior.

La primera propuesta de arquitectura alternativa a la original se denominó como pix2code poco profundo. Su arquitectura está basada en la del modelo pix2code (formulado en ecuaciones (4.3) - (4.6)). Sin embargo, el modelo de lenguaje consta solamente de una capa LSTM,

$$\mathbf{LM} = \text{LSTM}(\mathbf{y}, 128). \tag{4.7}$$

El resto del modelo y el entrenamiento de este es idéntico al original presentado en la Sección 4.3. En la Figura 4.2 se muestra la evolución de la

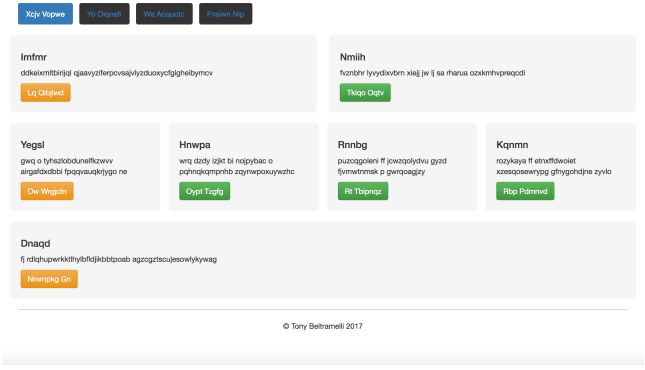
```
header {
btn-active, btn-inactive, btn-inactive, btn-inactive
}
row {
double {
small-title, text, btn-orange
}
double {
small-title, text, btn-green
}
}
row {
quadruple {
small-title, text, btn-orange
}
quadruple {
small-title, text, btn-green
}
quadruple {
small-title, text, btn-green
}
quadruple {
small-title, text, btn-green
}
}
row {
single {
small-title, text, btn-orange
}
}
```

(a) Secuencia de código objetivo

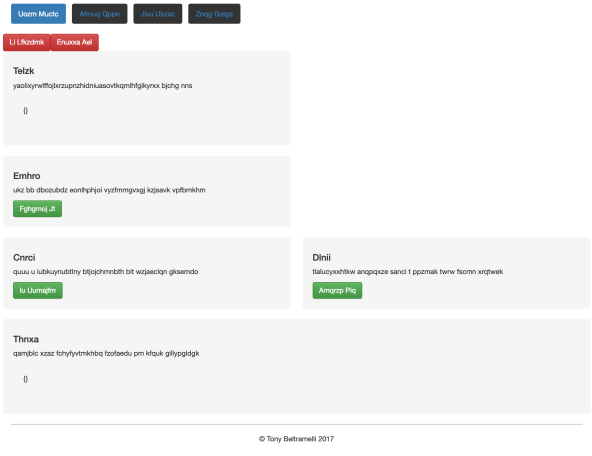
```
header{
btn-active,btn-inactive,btn-inactive,btn-inactive
}
row{
btn-red{
small-title,text,quadruple
}
btn-red{
small-title,text,btn-green
}
}
row{
double{
small-title,text,quadruple
}
double{
small-title,text,btn-green
}
double{
small-title,text,btn-green
}
double{
small-title,text,btn-green
}
}
row{
single{
small-title,text,quadruple
}
}
```

(b) Secuencia de código generada

Figura 4.3: Secuencias de código objetivo vs predicción



(a) Interfaz gráfica objetivo



(b) Interfaz gráfica predicha

Figura 4.4: Interfaz gráfica objetivo vs predicción

función de pérdida y métricas para el modelo original, en la Figura 4.5 para el modelo poco profundo y en la Figura 4.6 para el modelo con mecanismo de atención. El segundo modelo alternativo cuenta con un mecanismo de atención sobre la salida de la primera capa LSTM, en el modelo de lenguaje.

$$\mathbf{l} = \text{LSTM}(\mathbf{y}, 128), \quad \mathbf{att} = \text{Attention}(\mathbf{l}), \quad \mathbf{LM} = \text{LSTM}(\mathbf{att}, 128). \tag{4.8}$$

El mecanísmo de atención pondera elementos de la secuencia codificada a través de una regresión softmax. Este mecanísmo ha sido utilizado, por ejemplo en [Bahdanau et al., 2014], mostrando mejoras para tareas de traducción.

$$\text{Attention}(\mathbf{seq}) = \text{softmax}(\mathbf{seq}) \odot \mathbf{seq} \tag{4.9}$$

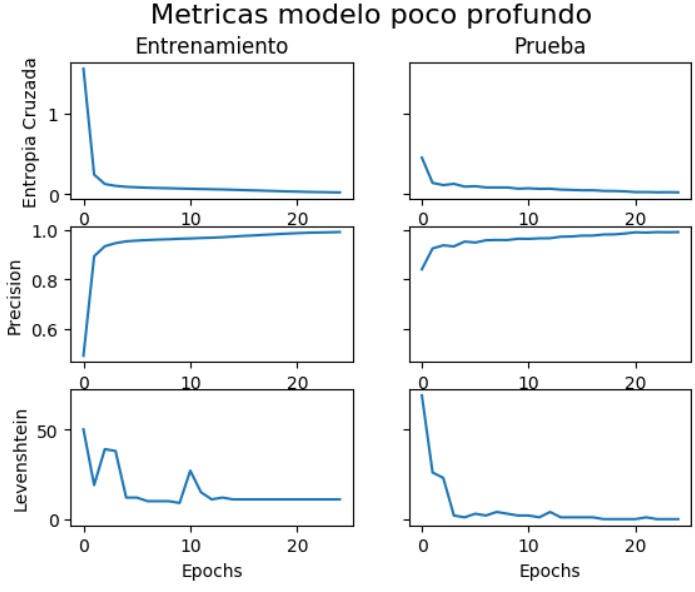


Figura 4.5: Métricas a través de las épocas modelo poco profundo

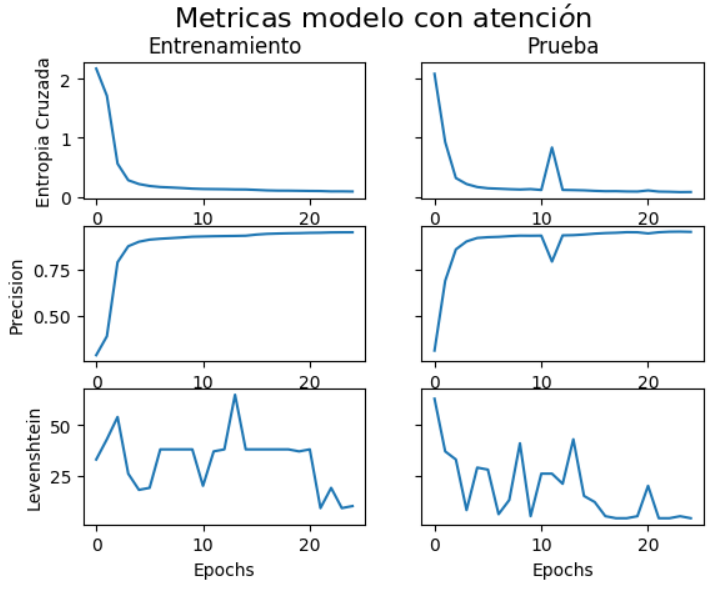


Figura 4.6: Métricas a través de las épocas modelo con atención

Notamos que el modelo poco profundo logra mejores niveles de entropía cruzada a través de las épocas. Esto es poco intuitivo ya que usualmente un modelo de mayor profundidad, al contar con más parámetros, puede ajustar de mejor forma los datos. Sin embargo, ajusta los datos de entrenamiento por lo que esta mejora puede venir entonces de un sobreajuste del modelo original a las secuencias de código. La segunda alternativa de arquitectura agrega una estructura, en vez de quitar como la alternativa anterior, agregando parámetros. Este modelo también presenta una mejora en la función de pérdida y en las métricas analizadas. Se hace referencia a la tabla 4.2 para granularidad. A pesar de ser un modelo con más parámetros, este captura de forma más eficiente los patrones en las secuencias procesadas,

evitando sobreajustar y obteniendo mejores métricas.

Con esto, se ha replicado parte del trabajo [Beltramelli, 2017] y propuesto alternativas para la solución de este problema. No solo se logró mapear datos de imágenes a texto, sino también capturar dependencias temporales en un lenguaje de programación. El Lenguaje de Dominio Específico usado tal vez sea de baja complejidad por lo que el modelo poco profundo logra capturar la estructura secuencial de manera adecuada. Sin embargo, como se expone en el siguiente capítulo, se podría intentar resolver el problema no reducido usando lenguajes de programación nativos.

Modelo	Conjunto	Entropía Cruzada	Precisión	Levenshtein
Original	Entrenamiento	0.14404	0.91654	40
Original	Prueba	0.13757	0.91753	9
Con Atención	Entrenamiento	0.08703	0.95194	10
Con Atención	Prueba	0.07868	0.95417	4
Poco Profundo	Entrenamiento	0.02034	0.99278	11
Poco Profundo	Prueba	0.02146	0.99268	0

Cuadro 4.2: Tamaño de conjuntos de entrenamiento y prueba

Capítulo 5

Conclusiones

En esta tesis se exploró el problema de generar programáticamente el código que da lugar a una interfaz gráfica. Este problema yace en el campo de estudio llamado programación inducida. Después de exponer la importancia de este campo así como el de Inteligencia Artificial, se hizo una breve descripción de los inicios de ambas y su desarrollo al día de hoy. Como mencionado anteriormente, el principal objeto de estudio de este trabajo se centra en la resolución del problema de generación de código. La solución explorada en este trabajo es la propuesta en [Beltramelli, 2017]. Esta hace uso de un modelo de aprendizaje profundo, una red neuronal.

Para comprender la arquitectura del modelo explorado se detalló la construcción matemática de sus dos componentes base, redes convolucionales y redes LSTM. La construcción de estas se motivo con un modelo lineal que busca encontrar el modelo generador usando el método Gradiente Estocástico. Tomando este algoritmo y el modelo lineal como punto de

partida, se construyó también un modelo lineal para clasificación. El problema de clasificación es de particular importancia dado que el problema de generar código se plantea como un problema de clasificación iterativo.

Se progresó de los modelos lineales a las redes neuronales al no poder modelar la función XOR con un modelo lineal de clasificación. Al no lograr estimar esta función linealmente, se expuso un modelo no lineal que lo logra, el perceptrón de una capa escondida. Partiendo una vez más de lo particular a lo general, se usó este modelo para presentar su generalización: el perceptrón de múltiples capas. Para la presentación de este modelo, se detalló su construcción desde su bloque elemental, la neurona. Una vez descrita la topología de estos modelos, se presentó el reto de calcular el gradiente de estos para ser usados durante entrenamiento usando Gradiente Estocástico. Particularmente, se expuso el método de propagación hacia atrás y el problema del gradiente desvaneciente/explotante. Temas de extrema relevancia para el entrenamiento exitoso de estos modelos.

Con conocimiento básico sobre redes neuronales, se expuso detalladamente dos arquitecturas distintas al peceptrón multicapas. Estas arquitecturas son eficientes respectivamente para el procesamiento de imágenes y secuencias de texto. Para caracterización de imágenes, se presentó la capa convolucional para redes neuronales usada ampliamente en visión computacional. Para modelar lenguaje, se describió la noción de recurrencia, una forma de modelarla con redes neuronales así como una estructura práctica, la capa LSTM. Una vez que se contó con el conocimiento necesario, se describió el modelo propuesto en [Beltramelli, 2017] para la resolución del problema principal de este trabajo. Finalmente, se propuso dos arquitec-

turas alternativas para la resolución de este. De hecho, estas alternativas logran mejores resultados que la original en las métricas analizadas. El hecho de poder resolver el problema con el modelo alternativo más simple que el original, prueba que las secuencias de código del LDE usado son suficientemente simples. Esto parece indicar que se podría tratar de resolver este problema con el modelo original, para secuencias de código en el lenguaje nativo. Esto se propone con mayor detalle en la sección 5.1.

Al termino de este trabajo, el lector cuenta con conocimiento sobre estructuras usadas para visión computacional y procesamiento de lenguaje. Aun más, tras explorar el problema de pix2code, el lector cuenta con una base de cómo usar ambas estructuras, tanto para visión como para lenguaje, para resolver problemas que hacen uso de ambos tipos de datos. Así, con un problema de clasificación se introduce al lector al campo de programación inducida. La exposición del lector a este campo abre un mar de posibilidades de problemas a ser resueltos con las estructuras y técnicas en aprendizaje profundo expuestas en este trabajo. A pesar de haber presentado una solución al problema principal, existen varias formas de continuar este trabajo.

5.1. Trabajo futuro

Extender este trabajo tiene varias alternativas, a continuación se proponen algunas opciones casi inmediatas. Recordamos que el código asociado a las interfaces gráficas es en realidad un Lenguaje de Dominio Específico. Se uso este LDE con la finalidad de reducir el espacio de bùsqueda del mo-

delo a optimizar. Dado que se cuenta con un modelo capaz de resolver el problema reducido, una extensión natural sería usar el lenguaje nativo de las interfaces. Probablemente esto implica que el espacio de búsqueda será considerablemente más grande y las estructuras sintácticas más complejas. Dada esta complejidad, es probable que el modelo que intente capturar dependencias temporales en el código necesite mejores capacidades de modelaje. Esto se podría ser lograr con más capas LSTM en el modelo de lenguaje o con un mecanismo de atención como el propuesto en la segunda arquitectura alternativa.

Bibliografía

Miltiadis Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 207–216. IEEE Press, 2013.

Marcin Andrychowicz and Karol Kurach. Learning efficient algorithms with hierarchical attentive memory. 2016.

Martin Arjovsky, Amar Shah, and Yoshua Bengio. Unitary evolution recurrent neural networks. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, pages 1120–1128. JMLR.org, 2016. URL http://dl.acm.org/citation.cfm?id=3045390.3045509.

Widrow B. An adaptive .ªdaline"neuron using chemical memistors. 1960.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

Tony Beltramelli. pix2code: Generating code from a graphical user interface screenshot. *arXiv preprint arXiv:1705.07962*, 2017.

Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994. URL http://www.iro.umontreal.ca/~lisa/pointeurs/ieeetrnn94.pdf.

Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. Statistical deobfuscation of android applications. In *Proceedings of the*

2016 ACM SIGSAC Conference on Computer and Communications Security, pages 343–355. ACM, 2016.

Allan W. Bierman. The inference of regular lisp programs from examples. *IEEE Transactions on Systems, Man, and Cybernetics*, 8(8):585–600, August 1978.

Léon Bottou, Frank E. Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning, 2016. URL https://arxiv.org/abs/1606.04838. quantization overview.

Dartmouth. Commemorating the 1956 founding at dartmouth college of ai as a research discipline, 2006. URL https://www.dartmouth.edu/~ai50/homepage.html.

Rosenblatt Frank. Principles of neurodynamics: Perceptrons and the theory of brain mechanisms. *Brain Theory*, pages 245–248, 1957.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. 2014.

Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Series in Statistics. Springer, 2da edition, 2009.

Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint*, 2012.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997. doi: 10.1162/neco.1997.9.8.1735. URL https://doi.org/10.1162/neco.1997.9.8.1735.

Armand Joulin and Tomas Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Advances in neural information processing systems*, pages 190–198, 2015.

Andrej Karpathy and Fei-Fei Li. Deep visual-semantic alignments for generating image descriptions. *CoRR*, abs/1412.2306, 2014. URL http://arxiv.org/abs/1412.2306.

Stephen Cole Kleene. *Introduction to Metamathematics*. North Holland, 1952.

Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010. URL http://yann.lecun.com/exdb/mnist/.

Zohar Manna and Richard J. Waldinger. Knowledge and reasoning in program synthesis. *Artificial Intelligence.*, 6(2):175–208, 1975.

John McCarthy and et al. A proposal for the dartmouth summer research project on artificial intelligence. *AI Magazine Volume 27 Number 4*, pages 12–14, 1955. URL https://www.aaai.org/ojs/index.php/aimagazine/article/view/1904/1802.

Warren McCulloch and Walter Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biology Vol 52*, pages 99–115, 1943.

Aditya Menon, Omer Tamuz, Sumit Gulwani, Butler Lampson, and Adam Kalai. A machine learning framework for programming by example. In *International Conference on Machine Learning*, pages 187–195, 2013.

Stephen Muggleton. Alan turing and the development of artificial intelligence. *AI Commun.*, 27(1):3–10, January 2014. ISSN 0921-7126. URL http://dl.acm.org/citation.cfm?id=2594611.2594613.

Arvind Neelakantan, Quoc V. Le, and Ilya Sustkever. Neural programmer: Inducing latent porgrams with gradient descent. 2016.

Peter Norvig and Robert Russel. *Artificial Inteligence: A Modern Approach*. Series in Artificial Intelligence. Prentice Hall, 2da edición edition, 2003.

Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Acm Sigplan Notices*, volume 49, pages 419–428. ACM, 2014.

Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from big code. In *ACM SIGPLAN Notices*, volume 50, pages 111–124. ACM, 2015.

Scott Reed and Nando de Freitas. Neural programmer-interpreters. 2016.

John Schulman, Nicolas Heess, Theophane Weber, and Pieter Abbeel. Gradient estimation using stochastic computation graphs. *CoRR*, abs/1506.05254, 2015. URL http://arxiv.org/abs/1506.05254.

Armando Solar-Lezama. The sketching approach to program synthesis. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems*, APLAS '09, pages 4–13, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-10671-2. doi: 10.1007/978-3-642-10672-9_3. URL http://dx.doi.org/10.1007/978-3-642-10672-9_3.

Phillip D. Summers. A methodology for lisp program construction from examples. *J. ACM*, 24(1):161–175, January 1977. ISSN 0004-5411. doi: 10.1145/321992.322002. URL http://doi.acm.org/10.1145/321992.322002.

Alan Turing. Computing machinery and intelligence. *Mind 49*, pages 433–460, 1950.

Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936. URL http://www.cs.helsinki.fi/u/gionis/cc05/OnComputableNumbers.pdf.

Richard J. Waldinger and Richard C. T. Lee. Prow: A step toward automatic program writing. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, IJCAI'69, pages 241–252, San Francisco, CA, USA, 1969. Morgan Kaufmann Publishers Inc. URL http://dl.acm.org/citation.cfm?id=1624562.1624586.

Jason Weston, Sumit Chopra, and Antoine Bordes. Memory networks. corr abs/1410.3916, 2014.

Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell:

Neural image caption generation with visual attention. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 2048–2057, Lille, France, 07–09 Jul 2015. PMLR. URL http://proceedings.mlr.press/v37/xuc15.html.