Daniel Voigt Godoy

Aprendizaje Profundo con PyTorch

Paso a Paso

Una Guía para Principiantes

Volumen I: Fundamentos

Traducción de Jesús Martínez-Blanco

Aprendizaje Profundo con PyTorch Paso a Paso: Una Guía para Principiantes

Volumen I—Fundamentos

Daniel Voigt Godoy

Version 1.1.1

Aprendizaje Profundo con PyTorch Paso a Paso: Una Guía para Principiantes

Volumen I—Fundamentos

por Daniel Voigt Godoy (Traducción de Jesús Martínez-Blanco)

Agosto 2022: Primera edición

Para obtener más información, envíe un correo electrónico a contact@dvgodoy.com

"*Lo que no puedo crear, no lo entiendo.*"

Richard P. Feynman

Tabla de Contenido

Prefacio

Si estás leyendo esto, probablemente no necesito decirte que el aprendizaje profundo (*deep learning*) es increíble y PyTorch es una librería genial, ¿verdad?

Pero déjame que te cuente brevemente cómo surgió la idea de escribir esta serie de libros. En 2016 comencé a impartir una clase sobre aprendizaje automático (machine learning) con Apache Spark y, un par de años más tarde, otra clase sobre los fundamentos de aprendizaje automático.

Poco tiempo después andaba buscando una entrada de blog que explicara visualmente, de una manera clara y concisa, el concepto de entropía cruzada binaria (binary cross-entropy) para poder explicarla a mis estudiantes. Como no pude encontrar ninguna que fuera adecuada, decidí escribir una yo mismo. Aunque pensé que era un tema bastante básico, resultó ser mi artículo más popular[1]. Mis lectores han dado la bienvenida a la manera sencilla, directa y llana en que expliqué el tema.

Más tade, en 2019, usé el mismo enfoque para escribir otra publicación: "Understanding PyTorch with an example: a step-by-step tutorial."[2] Una vez más, ¡me sorprendió la reacción de los lectores!

Fueron sus comentarios positivos los que me motivaron a escribir esta serie de libros para ayudar a los principiantes a iniciar su viaje hacia el aprendizaje profundo y PyTorch..

En este primer volumen, cubro los fundamentos del descenso de gradiente (gradient descent), los fundamentos de PyTorch, las regresiones lineales y logísticas, las métricas de evaluación etc. Si no tienes absolutamente ninguna experiencia con PyTorch, verás que este libro es tu punto de partida.

El segundo volumen se centra principalmente en la visión artificial (computer vision): modelos más profundos y funciones de activación, redes neuronales convolucionales, esquemas de inicialización, schedulers y aprendizaje de transferencia. Si tu objetivo es aprender sobre modelos de aprendizaje profundo para la visión artificial, y ya te sientes cómodo entrenando modelos simples en PyTorch, el segundo volumen es el adecuado para ti.

Luego, el tercer volumen se centra en todas las cosas relacionadas con secuencias: redes neuronales recurrentes y sus variaciones, modelos de secuencia a secuencia, atención, autoatención y la arquitectura del Transformer. El último capítulo del

tercer volumen es un curso intensivo sobre el procesamiento del lenguaje natural (natural language processing): desde los conceptos básicos de la tokenización de palabras hasta el ajuste de modelos grandes (BERT y GPT-2) utilizando la librería HuggingFace. Este último volumen es más exigente que los otros dos, y vas a disfrutarlo más si ya tienes una comprensión sólida de los modelos de aprendizaje profundo.

Estos libros están diseñados para ser leídos en orden y, aunque *pueden* ser leídos de forma independiente, te recomiendo encarecidamente que los leas de forma secuencial, como si fuera un único libro :-)

Espero que disfrutes leyendo esta serie tanto como yo disfruté escribiéndola.

[1] https://bit.ly/2UW5iTg

[2] https://bit.ly/2TpzwxR

Agradecimientos

En primer lugar, me gustaría darte las GRACIAS a ti, mi lector, por hacer posible este libro. Si no fuera por los increíbles comentarios que recibí de los miles de lectores de mi blog sobre PyTorch, ¡nunca habría reunido la fuerza para comenzar y *terminar* una empresa tan importante como escribir una serie de 1,000 páginas!

Me gustaría agradecer a mis buenos amigos Jesús Martínez-Blanco (quien logró leer absolutamente todo lo que escribí), Jakub Cieslik, Hannah Berscheid, Mihail Vieru, Ramona Theresa Steck, Mehdi Belayet Lincon y António Góis por ayudarme y dedicar una buena parte de su tiempo a leer, corregir y sugerir mejoras a mis borradores. ¡Estaré siempre agradecido por vuestro apoyo! También me gustaría agradecer a mi amigo José Luis López Pino por el empujón inicial que necesitaba para realmente *empezar* a escribir este libro.

Muchas gracias a mis amigos José Quesada y David Anderson por aceptarme como estudiante en el bootcamp *Data Science Retreat* en 2015 y, más tarde, por invitarme a ser profesor allí. Ese fue el punto de partida de mi carrera como científico de datos y como profesor.

También me gustaría agradecer a los desarrolladores de PyTorch por desarrollar una librería tan increíble, y a los equipos de Leanpub y Towards Data Science por hacer que sea increíblemente fácil para los creadores de contenido como yo compartir su trabajo con la comunidad.

Por último, me gustaría dar las gracias a mi esposa, Jerusa, por ser siempre mi apoyo durante toda la escritura de esta serie de libros, y por tomarse el tiempo para leer *cada página* de la misma :-)

Acerca de los autores

Daniel es un científico de datos, desarrollador, escritor y profesor. Ha estado enseñando aprendizaje automático y tecnologías de la computación distribuida en Data Science Retreat, el bootcamp con sede en Berlín más antiguo, desde 2016, ayudando a más de 150 estudiantes a progresar en sus carreras como científicos de datos. Su trayectoria profesional incluye 20 años de experiencia trabajando para empresas de varios sectores: banca, gobierno, fintech, venta al por menor y movilidad.

Jesús es físico y científico de datos. Es también profesor y mentor en Data Science Retreat, donde imparte cursos de Visualización de Datos en el contexto web. Desde 2016 lleva desarrollando múltiples productos de automatización en los sectores de publicidad online y movilidad.

Preguntas frecuentes

¿Por qué PyTorch?

En primer lugar, programar en PyTorch es **divertido** :-) En serio, tiene algo que hace que sea agradable escribir código. Hay quien dice que es porque es muy **pythonic**, o tal vez hay algo más, ¿quién sabe? ¡Espero que, al final de este libro, tú también lo veas así! En segundo lugar, puede que haya incluso algunos *beneficios inesperados* para tu salud, como dice Andrej Karpathy en este tweet[3].

Bromas aparte, PyTorch es la biblioteca de aprendizaje profundo que **más ha crecido**[4] y existe un **enorme ecosistema**.[5] de *herramientas* y *bibliotecas* desarrolladas en base a PyTorch. Es ya la **biblioteca preferida**[6] en el mundo académico y está abriéndose rápidamente camino en la industria.

Varias empresas utilizan ya el poder de PyTorch;[7], como por ejemplo:

- **Facebook**: Esta compañía es la desarrolladora original de PyTorch, lanzado en octubre de 2016.
- **Tesla**: Mira a Andrej Karpathy (director de IA en Tesla) hablar sobre "*cómo Tesla está usando PyTorch para desarrollar capacidades completas de autoconducción para sus vehículos*" en este video.[8]
- **OpenAI**: En enero de 2020, OpenAI decidió estandarizar su marco de aprendizaje profundo en PyTorch.[9]
- **fastai**: fastai es una biblioteca[10] basada en PyTorch para simplificar el entrenamiento de modelos y se utiliza en su curso *"Practical Deep Learning for Coders"*[11]. La biblioteca fastai está estrechamente relacionada con PyTorch y "*no puedes ser realmente competente en el uso de fastai si no conoces bien PyTorch*."[12]
- **Uber**: La compañía es un contribuyente significativo al ecosistema de PyTorch, habiendo desarrollado bibliotecas como Pyro[13] (programación probabilística) y Horovod[14] (un marco de entrenamiento de modelos distribuido).
- **Airbnb**: PyTorch constituye el alma de su asistente de diálogo para el servicio al cliente.[15]

Esta serie de libros **tiene como objetivo introducirte en el mundo PyTorch** a la vez que te da una **comprensión sólida de cómo funciona**.

¿Por qué este libro?

Si estás buscando un libro donde puedas aprender sobre el aprendizaje profundo y PyTorch sin tener que pasar horas descifrando texto y código críptico, si buscas un libro que sea además fácil y agradable de leer, aquí lo tienes :-)

En primer lugar, éste **no** es el típico libro que *empieza* con algún sencillo *problema de clasificación de imágenes* para ilustrar cómo usar PyTorch. Puede parecer una buena idea, pero creo que **te distrae** del **objetivo principal**: aprender **cómo funciona PyTorch**. En este libro presento un enfoque **estructurado**, **incremental** y **a partir de primeros principios** para aprender PyTorch.

En segundo lugar, éste **no** es en absoluto un **libro formal**: escribo este libro **como si estuviera teniendo una conversación contigo**, el lector. Te haré **preguntas** (y te daré las respuestas poco después), y también haré (en ocasiones absurdos) **chistes**.

Mi trabajo aquí es hacerte **entender** los temas que vamos a abordar, así que **evitaré notación matemática sofisticada** tanto como sea posible e intentaré **usar un español llano**.

En este primer libro de la serie *Aprendizaje Profundo con PyTorch Paso a Paso*, te **guiaré** a través del **desarrollo** de muchos modelos en PyTorch, mostrándote por qué PyTorch hace que sea mucho **más fácil** y más **intuitivo** construir modelos en Python: *autograd*, *grafo computacional dinámico*, *clases* del modelo, y mucho, mucho más.

Vamos a construir, **paso a paso**, no solo los modelos en sí, sino también tu **comprensión** de los mismos, ya que te mostraré el **razonamiento** que hay trás el código y **cómo evitar** algunos **escollos y errores comunes** por el camino.

Hay otra ventaja de **centrarse en lo básico**: es probable que este libro tenga por ello una **vida útil más larga**. Es bastante común que los libros técnicos, especialmente los que se centran en tecnologías de vanguardia, se vuelvan obsoletos rápidamente. Esperemos que este no sea el caso aquí, ya que tanto la **mecánica subyacente** como los **conceptos** que la sustentan, no van a cambiar. Es esperable que la sintaxis de algunos ejemplos de programación cambien con el tiempo, pero no creo que esos cambios sean relevantes al menos para el futuro inmediato.

Una cosa más: Si aún no te habías dado cuenta, **me encanta** hacer uso de **señales visuales**, es decir, destacar el texto con **negrita** y *cursiva*. Creo firmemente que esto ayuda al lector a **comprender** más fácilmente las **ideas clave** que estoy intentando transmitir. Puedes encontrar más sobre esto en la sección "**Cómo leer este libro.**"

¿Quién debería leer este libro?

Escribí este libro para **principiantes en general**, no solo para principiantes en PyTorch. De vez en cuando dedicaré algún tiempo a explicar algunos **conceptos fundamentales** que creo que son **esenciales** para tener una comprensión adecuada **del funcionamiento del código**.

El mejor ejemplo es el **descenso de gradiente** (gradient descent), con el que la mayoría de las personas están familiarizadas en mayor o menor medida. Tal vez sepas cuál es la idea general, tal vez la hayas visto en el curso de aprendizaje automático de Andrew Ng, o tal vez incluso hayas **calculado algunas derivadas parciales tú mismo**.

En la vida real, la **mecánica** del descenso de gradiente será **responsabilidad de PyTorch** (¡alerta de spoiler!). En cualquier caso, intentaré darte las claves de su funcionamiento (a menos que elijas omitir el Capítulo 0 por completo, por supuesto), porque muchos **elementos en el código**, así como **opciones de hiperparámetros** (tasa de aprendizaje, tamaño de mini-batch, etc.), se pueden entender mucho más fácilmente si sabes **de dónde vienen**.

Tal vez ya conozcas bien algunos de estos conceptos: Si es así, puedes simplemente *saltártelos*, ya que he procurado que estas explicaciones sean lo más *independientes* del resto del contenido como sea posible en cada caso.

Pero **quiero asegurarme de que todos estamos al mismo nivel de entendimiento**, por lo que, si es la primera vez que escuchas sobre un concepto o si no estás seguro de haberlo entendido por completo, estas explicaciones son para ti.

¿Qué necesito saber?

Este es un libro para principiantes, por lo que asumo un **conocimiento previo** tan mínimo como sea posible; como ya dije en la sección anterior, me tomaré el tiempo para explicar conceptos fundamentales siempre que sea necesario.

Dicho esto, aquí va una lista de lo que espero de ti, el lector:

- ser capaz de programar en **Python** (si estás familiarizado con la programación orientada a objetos [OOP por sus siglas en inglés], incluso mejor)
- ser capaz de utilizar las bibliotecas de PyData (**numpy**, **matplotplib** y **pandas**) y los **Jupyter Notebooks**
- estar familiarizado con algunos conceptos básicos utilizados en **aprendizaje automático**, como:
  - aprendizaje supervisado: regresión y clasificación
  - funciones error para problemas de regresión y clasificación (error cuadrático medio, entropía cruzada, etc.)
  - división del conjunto de datos en entrenamiento, validación y test
  - subajuste y sobreajuste (dilema sesgo/varianza)

En cualquier caso, trataré **algunos** de estos temas solo de forma breve, de lo contrario, ¡este libro sería gigantesco!

Cómo leer este libro

Como este libro es una **guía para principiantes**, está diseñado para ser leído **secuencialmente**, al tiempo que las ideas y los conceptos se van construyendo progresivamente. Lo mismo ocurre con el **código** dentro del libro: deberías poder *reproducir* todos los resultados, siempre que ejecutes los fragmentos de código en el mismo orden en que se introducen.

Este libro es **visualmente** diferente a otros libros: Como ya he mencionado en la sección "**¿Por qué este libro?**", me gusta **mucho** usar **señales visuales**. Aunque esto no es, *estrictamente hablando*, una **convención**, así es como puedes interpretar esas señales:

- Utilizo **negrita** para resaltar lo que creo que son las **palabras más relevantes** en una frase o párrafo, mientras que las palabras *en cursiva* también se consideran *importantes* (aunque no lo suficientemente importantes como para estar en negrita)
- *Variables*, *coeficientes*, y *parámetros* en general, están *en cursiva*
- `Clases` y `métodos` están escritos en una fuente `monoespaciada`, y enlazan a la documentación PyTorch[16] la primera vez que se introducen, para que puedas

encontrarlo fácilmente (a diferencia de otros enlaces en este libro, los enlaces a la documentación son *numerosos* y por lo tanto *no* incluidos en las notas a pie de página)

- Cada **celda de código** va seguida de *otra* celda que muestra los **resultados** correspondientes (si las hubiera)
- **Todo el código** presente en este libro está disponible en el **repositorio oficial** de GitHub:

https://github.com/dvgodoy/PyTorchStepByStep

Las celdas de código con **títulos** son una parte importante del flujo de trabajo:

El Título va aquí

```
# Lo que quiera que vaya aquí, va a tener un impacto en OTRAS
# celdas. Además, la mayoría de las celdas tienen COMENTARIOS
# que explican por dónde van los tiros
x = [1., 2., 3.]
print(x)
```

Si hay algún resultado de la celda de código, con título o sin él, *habrá* otra celda de código que represente el **resultado** correspondiente para que puedas *verificar* si lo has reproducido con éxito o no.

Output

```
[1.0, 2.0, 3.0]
```

Algunas celdas de código **no** tienen títulos, su ejecución no afecta al flujo de trabajo:

```
# Esas celdas ilustran CÓMO PROGRAMAR algo pero
# NO son parte del flujo de trabajo principal
dummy = ['a', 'b', 'c']
print(dummy[::-1])
```

¡Pero incluso estas celdas muestran sus resultados!

Output

```
['c', 'b', 'a']
```

Utilizo textos al margen para comunicar diferentes cosas, dependiendo del icono correspondiente:

ATENCIÓN

Algún **problema** en potencia o alguna cosa de la que debamos estar **atentos**.

CONSEJO

Aspectos clave que realmente quiero que **recuerdes**.

INFORMACIÓN

Información importante a la que **prestar atención**.

IMPORTANTE

Información a la que **realmente prestar atención**.

TÉCNICO

Aspectos técnicos de **parametrización** o **funcionamiento interno de algoritmos**.

PREGUNTA Y RESPUESTA

Haciéndome **preguntas** (pretendiendo ser tú, el lector) y respondiéndolas, ya sea en el mismo bloque o poco después.

DISCUSIÓN

Discusión realmente breve sobre un concepto o tema.

DESPUÉS

Temas importantes que se tratarán con más detalle más adelante.

BROMA

Chistes, juegos de palabras, memes, citas de películas.

¿Qué es lo siguiente?

Es hora de **configurar** un entorno para tu aventura de aprendizaje utilizando la **Guía de configuración**.

[3] https://bit.ly/2MQoYRo
[4] https://bit.ly/37uZgLB
[5] https://pytorch.org/ecosystem/
[6] https://bit.ly/2MTN0Lh
[7] https://bit.ly/2UFHFve
[8] https://bit.ly/2XXJkyo
[9] https://openai.com/blog/openai-pytorch/
[10] https://docs.fast.ai/
[11] https://course.fast.ai/
[12] https://course.fast.ai/
[13] http://pyro.ai/
[14] https://github.com/horovod/horovod
[15] https://bit.ly/30CPhm5
[16] https://bit.ly/3cT1aH2

Guía de configuración

Repositorio oficial

El repositorio oficial de este libro está disponible en GitHub:

https://github.com/dvgodoy/PyTorchStepByStep

Contiene **un Jupyter notebook** por cada **capítulo** de este libro. Cada notebook contiene **todo el código presente** en el capítulo correspondiente, y deberías poder **ejecutar sus celdas secuencialmente** para obtener los **mismos resultados (*outputs*)** que se muestran en el libro. Creo firmemente que ser capaz de **reproducir los resultados** infunde **confianza** en el lector.

A pesar de que hice todo lo posible para garantizar la **reproducibilidad** de los resultados, es posible que encuentres **aún** algunas diferencias menores en tus resultados (especialmente durante el entrenamiento de los modelos). Desgraciadamente, los resultados completamente reproducibles no están garantizados en todas las versiones de PyTorch, y los resultados pueden no ser reproducibles entre ejecuciones usando la CPU y la GPU, incluso cuando se usan *seeds* idénticas.[17]

Entornos de programación

Hay **tres opciones** para que ejecutes los Jupyter notebooks:

- Google Colab (*https://colab.research.google.com*)
- Binder (*https://mybinder.org*)
- Instalación local

Exploremos brevemente los **pros** y **contras** de cada una de estas opciones.

Google Colab

Google Colab "*te permite escribir y ejecutar Python en tu navegador, sin necesidad de configuración, acceso gratuito a GPU y la posibilidad de compartir tu código de forma fácil.*"[18].

Puedes **cargar los notebooks directamente desde GitHub** con la URL especial de Colab (*https://colab.research.google.com/github/*). Simplemente escribe el usuario u organización de GitHub (como el mío, `dvgodoy`), y te mostrará una lista de todos sus repositorios públicos (como la de este libro, `PyTorchStepByStep`).

Después de elegir un repositorio, se te mostrará una lista de los notebooks disponibles y los enlaces correspondientes para abrirlos en una nueva pestaña del navegador.

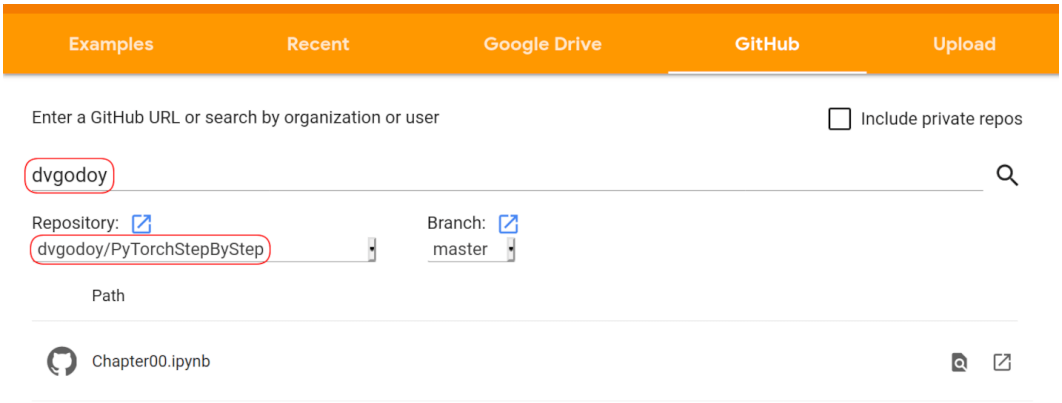

Figura S.1 - La URL especial de Google Colab

También tienes acceso a una **GPU**, que es muy útil para entrenar modelos de aprendizaje profundo **más rápido**. Más importante aún, si **realizas cambios** en el notebook, Google Colab **los conservará**. Esta forma de trabajar me parece muy conveniente; las únicas **desventajas** que veo son:

- Tienes que **haber iniciado sesión** en una cuenta de Google.
- Necesitas (re)instalar paquetes de Python que no forman parte de la configuración predeterminada de Google Colab.

Binder

Binder "*te permite crear entornos personalizados de programación que pueden ser compartidos y utilizados por muchos usuarios remotos.*"[19]

También puedes **cargar notebooks directamente desde GitHub**, pero el proceso es ligeramente diferente. Binder creará algo así como una *máquina virtual* (técnicamente, es un contenedor, pero dejémoslo ahí), clonará el repositorio e iniciará Jupyter. Esto te permite tener acceso a **la página de inicio de Jupyter** en tu navegador, igual que lo harías si lo estuvieras ejecutando localmente, pero todo se está ejecutando en un servidor JupyterHub.

Simplemente ve a Binder (*https://mybinder.org/*) y escribe la URL del repositorio de GitHub que quieras explorar (por ejemplo, `https://github.com/dvgodoy/PyTorchStepByStep`) y haz clic en **Launch** (*Iniciar*). Tardará un par de minutos en crear la imagen y abrir la página de inicio de Jupyter.

También puedes **iniciar Binder** para el repositorio de este libro directamente utilizando el siguiente enlace: *https://mybinder.org/v2/gh/dvgodoy/PyTorchStepByStep/master*.

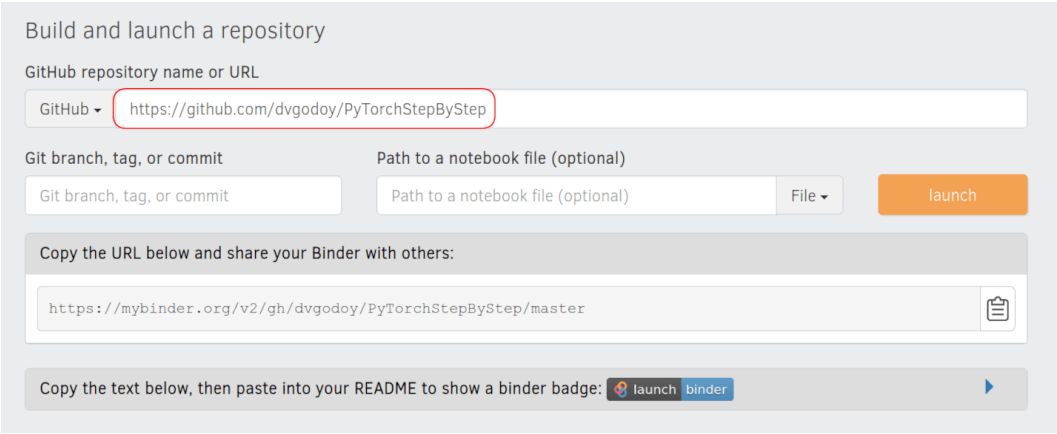

Figura S.2 - La página de Binder

Binder es muy conveniente ya que **no requiere de ningún tipo de configuración previa**. Es probable que los paquetes de Python necesarios para ejecutar correctamente el entorno se instalen durante el inicio (si lo proporciona el autor del repositorio).

Por otro lado, arrancar puede **tardar tiempo**, y **no mantiene tus cambios** después de que expire tu sesión (por lo tanto, asegúrese de **descargar** cualquier notebook que modifiques).

Instalación local

Esta opción te dará más **flexibilidad**, pero requerirá **más esfuerzo para configurarla**. Te animo a que intentes crear tu propio entorno de programación. Puede parecer desalentador al principio, pero seguramente puedes lograrlo si sigues estos **siete pasos sencillos**:

Lista de pasos

- [ ] 1. Instala **Anaconda**.
- [ ] 2. Crea y activa un **entorno virtual**.
- [ ] 3. Instala el paquete **PyTorch**.
- [ ] 4. Instala el paquete **TensorBoard**.
- [ ] 5. Instala el software **GraphViz** y el paquete **TorchViz** (**opcional**).
- [ ] 6. Instala **git** y **clona** el repositorio.
- [ ] 7. Inicia el **Jupyter** notebook.

1. Anaconda

Si aún no tienes instalada **Anaconda's Individual Edition**[20], este sería un buen momento para hacerlo. Es una forma conveniente de comenzar, ya que contiene la mayoría de los paquetes de Python que un científico de datos necesitará para desarrollar y entrenar modelos.

Sigue las **instrucciones de instalación** correspondientes a tu sistema operativo:

- Windows (*https://docs.anaconda.com/anaconda/install/windows/*)
- macOS (*https://docs.anaconda.com/anaconda/install/mac-os/*)
- Linux (*https://docs.anaconda.com/anaconda/install/linux/*)

Asegúrate de elegir la versión **Python 3.X**, ya que se ha dejado de mantener Python 2 desde enero de 2020.

Después de instalar Anaconda, es hora de crear un **entorno**.

2. Entornos (virtuales) Conda

Los entornos virtuales son una forma muy conveniente de aislar las instalaciones de Python asociadas con diferentes proyectos.

"*¿Qué es un entorno?*"

Es más o menos una **réplica del propio Python y algunas (o todas) de sus bibliotecas**, por lo que, efectivamente, terminarás teniendo múltiples instalaciones de Python en tu ordenador.

"*¿Por qué no puedo usar una sola instalación de Python para todo?*"

Con tantas **bibliotecas** desarrolladas de forma independiente, cada una con muchas **versiones** diferentes y cada versión con varias **dependencias** (de otras bibliotecas), **las cosas pueden salirse de control** muy rápido.

No nos vamos a parar a debatir estos temas, pero puedes fiarte de mi palabra (¡o busca en Google!): terminará yéndote mucho mejor si adquieres el hábito de **crear un entorno diferente para cada proyecto en el que comiences a trabajar**.

"*¿Cómo creo un entorno?*"

Primero, debes elegir un **nombre** para tu entorno :-) Llamemos al nuestro `pytorchbook` (o cualquier otra cosa que te resulte fácil de recordar). A continuación, deberás abrir una ventana de **Terminal** (en Ubuntu) o **Anaconda Prompt** (en Windows o macOS) y escribir el siguiente comando:

```
$ conda create -n pytorchbook anaconda
```

El comando anterior crea un entorno de Conda llamado `pytorchbook` e incluye **todas los paquetes de Anaconda** (puedes ir a tomarte un café, tardará un tiempo…). Si deseas obtener más información sobre cómo crear y usar los entornos de Conda, puedes consultar "Managing Environments"[21] de la guía del usuario.

¿Se terminó de crear el entorno? ¡Bien! Es hora de **activarlo**, lo que significa que **esa instalación de Python** es la que se va a usar ahora. En la misma ventana del terminal (o prompt de Anaconda), simplemente escribe:

```
$ conda activate pytorchbook
```

El prompt debería tener este aspecto (si usas Linux):

```
(pytorchbook)$
```

o así (si está usando Windows):

```
(pytorchbook)C:\>
```

¡Listo! Está utilizando un **nuevo entorno de Conda** ahora. Necesitarás **activarlo** cada vez que abras un nuevo terminal, o, si eres usuario de Windows o macOS, puedes abrir el prompt correspondiente de Anaconda (aparecerá como **Anaconda Prompt (pytorchbook)**, en nuestro caso), que lo tendrá activado desde el principio.

IMPORTANTE: A partir de ahora, asumo que activarás el entorno `pytorchbook` cada vez que abras un terminal o un prompt de Anaconda. Cualquier paso subsiguiente de instalación **debe** ejecutarse dentro del entorno.

3. PyTorch

PyTorch es el mejor **marco de aprendizaje profundo** que existe, por si te saltaste la introducción.

Es "*un marco de aprendizaje automático de código abierto que reduce la distancia entre los primeros prototipos creados al inicio de una investigación y la implementación de una solución en producción.*"[22] No suena nada mal, ¿verdad? Bueno, probablemente ya no tengo que convencerte sobre este extremo :-)

Es hora de instalar la biblioteca protagonista de este libro :-) Podemos ir directamente a la sección **Start Locally** (*https://pytorch.org/get-started/locally/*) del sitio web de PyTorch, y automáticamente seleccionará las opciones que mejor se adapten a tu entorno local, y te mostrará el **comando que debes ejecutar**.

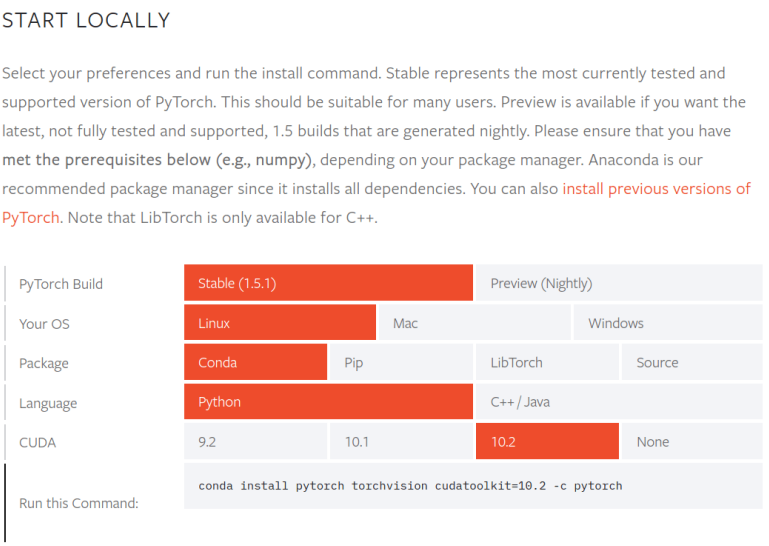

Figura S.3 - La página Start Locally de PyTorch

Algunas de estas opciones son:

- PyTorch Build: selecciona siempre una versión **estable**.
- Paquete: Asumo que estás usando **Conda**.
- Language: Obviamente, **Python**.

Solo quedan dos opciones: **Your OS** (tu sistema operativo) y **CUDA**.

"*¿Qué es CUDA?*", te preguntás.

Usar GPU / CUDA

CUDA "*es una plataforma de computación en paralelo y un modelo de programación desarrollado por NVIDIA para la computación general en Unidades de Procesamiento Gráfico (GPU por sus siglas en inglés).*"[23]

Si tienes una **GPU** en tu ordenador (probablemente una tarjeta gráfica *GeForce*), puedes aprovechar su poder para entrenar modelos de aprendizaje profundo **mucho más rápido** que usando una CPU. En este caso, debes elegir una instalación de PyTorch que sea compatible con CUDA.

Sin embargo, esto no es suficiente: si aún no lo has hecho, necesitas instalar controladores actualizados, el CUDA Toolkit y la biblioteca CUDA Deep Neural Network (cuDNN). Desgraciadamente, indagar en unas instrucciones de instalación más detalladas para CUDA están fuera del alcance de este libro.

La **ventaja** del uso de una GPU es que te permite **iterar más rápido** y **experimentar con modelos más complejos y una gama más amplia de hiperparámetros**.

En mi caso, uso **Linux**, y tengo una **GPU** con CUDA versión 10.2 instalada. Así que puedo ejecutar el siguiente comando en el **terminal** (después de activar el entorno):

```
(pytorchbook)$ conda install pytorch torchvision\
cudatoolkit=10.2 -c pytorch
```

Usar CPU

Si **no** tienes una **GPU**, debes elegir **None** para CUDA.

"*¿Puedo ejecutar el código **sin** una GPU*?", te preguntarás.

¡Claro! El código y los ejemplos de este libro fueron diseñados para que **todos los lectores** puedan seguirlos sin problemas. Puede que algunos ejemplos exijan un poco más de potencia computacional, pero estamos hablando de un **par de minutos** en una CPU, no de horas. Si no tienes una GPU, **¡no te preocupes**! Además, siempre puedes usar Google Colab si necesitas usar una GPU durante un tiempo.

Si tienes un **ordenador con Windows** y **sin GPU**, tienes que ejecutar el siguiente comando en el **prompt de Anaconda (pytorchbook)**:

```
(pytorchbook) C:\> conda install pytorch torchvision cpuonly\
 -c pytorch
```

Instalar CUDA

CUDA: Instalar controladores para una tarjeta gráfica GeForce, cuDNN de NVIDIA y el CUDA Toolkit puede ser bastante complicado y depende en gran medida del modelo de tarjeta que tengas y de tu sistema operativo.

Para instalar los controladores de GeForce, ve al sitio web de GeForce (*https://www.geforce.com/drivers*), selecciona tu sistema operativo y el modelo de tu tarjeta gráfica, y sigue las instrucciones de instalación.

Para instalar la biblioteca CUDA Deep Neural Network (cuDNN) de NVIDIA, debes registrarte en *https://developer.nvidia.com/cudnn*.

Para instalar el CUDA Toolkit (*https://developer.nvidia.com/cuda-toolkit*), sigue las instrucciones para tu sistema operativo y elije un instalador local o un archivo ejecutable.

macOS: Si eres usuario de macOS, ten en cuenta que los binarios de PyTorch **NO** son compatibles con **CUDA**, lo que significa que necesitarás instalar PyTorch **desde la fuente** si quieres usar tu GPU. Este es un proceso algo **complicado** (como se describe en *https://github.com/pytorch/pytorch#from-source*), así que si no tienes ganas de hacerlo, puedes elegir continuar **sin CUDA**, y aún así podrás ejecutar el código presente en este libro de forma rápida.

4. TensorBoard

TensorBoard es el **kit de herramientas de visualización** de TensorFlow, y "*proporciona la visualización y las herramientas necesarias para la experimentación con aprendizaje automático.*"[24]

TensorBoard es una herramienta poderosa, y podemos usarla incluso si estamos desarrollando modelos en PyTorch. Afortunadamente, no es necesario instalar toda la biblioteca TensorFlow para obtenerlo; puedes fácilmente **solo instalar TensorBoard** usando **Conda**. Solo necesitas ejecutar este comando en tu **terminal** o **prompt de Anaconda** (de nuevo, después de activar el entorno):

```
(pytorchbook)$ conda install -c conda-forge tensorboard
```

5. GraphViz y Torchviz (opcional)

Este paso es opcional, principalmente porque la instalación de GraphViz a veces puede ser *problemática* (especialmente en Windows). Si por alguna razón no logras instalarlo correctamente, o si decides omitir este paso de instalación, aún podrás **ejecutar el código presente en este libro** (salvo por un par de celdas que generan imágenes de la estructura de un modelo en la sección "Grafo Computacional Dinámico" del Capítulo 1).

GraphViz es un software de visualización de grafos de código abierto. Es "*una forma de representar información estructural como diagramas de grafos abstractos y redes.*"[25]

Necesitamos instalar GraphViz para usar **TorchViz**, un elegante paquete que nos permitirá visualizar la estructura de un modelo. Consulta las **instrucciones de instalación** para tu sistema operativo en *https://www.graphviz.org/download/*.

Si usas Windows, utiliza el instalador **GraphViz 's Windows Package** que puedes encontrar en *https://graphviz.gitlab.io/_pages/Download/windows/graphviz-2.38.msi*.

También es necesario añadir GraphViz en tu PATH (variable de entorno) en Windows. Lo más probable es que puedas encontrar el archivo ejecutable de GraphViz en `C:\ProgramFiles(x86)\Graphviz2.38\bin`. Una vez que lo encuentres, debes establecer o cambiar el PATH, añadiendo la ubicación de GraphViz en él.

Para obtener más detalles sobre cómo hacerlo, consulta "How to Add to Windows PATH Environment Variable."[26]

Para obtener información adicional, también puedes consultar la guía "How to Install Graphviz Software"[27].

Después de instalar GraphViz, puedes instalar el paquete **torchviz**[28]. Este paquete **no** es parte del Anaconda Distribution Repository[29] y solo está disponible en **PyPI**[30], el Índice de Paquetes de Python, por lo que necesitamos usar `pip install` para instalarlo.

Una vez más, abre una ventana de **terminal** o un **prompt de Anaconda** y ejecuta este comando (recuerda, una vez más: después de activar el entorno):

```
(pytorchbook)$ pip install torchviz
```

Para comprobar tu instalación de GraphViz /TorchViz, puedes probar el siguiente código de Python:

```
(pytorchbook)$ python

Python 3.7.5 (default, Oct 25 2019, 15:51:11)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>> import torch
>>> from torchviz import make_dot
>>> v = torch.tensor(1.0, requires_grad=True)
>>> make_dot(v)
```

Si todo **funciona correctamente**, deberías ver algo como esto:

Output

```
<graphviz.dot.Digraph object at 0x7ff540c56f50>
```

Si te da un **error** de cualquier tipo (el que se muestra a continuación es bastante común), significa que todavía hay algún tipo de **problema de instalación** con GraphViz.

Output

```
ExecutableNotFound: failed to execute ['dot', '-Tsvg'], make
sure the Graphviz executables are on your systems' PATH
```

6. Git

Está *mucho* más allá del alcance de esta guía enseñarte cómo funciona el sistema de control de versiones (version control) y su herramienta más popular: `git`. Si ya estás familiarizado con él, ¡genial, puedes saltarte esta sección por completo!

conda install vs pip install

Aunque puedan parecer equivalentes a primera vista, es **preferible usar `conda install`** en lugar de `pip install` cuando trabajas con Anaconda y sus entornos virtuales.

Esto se debe a que `conda install` es sensible al entorno virtual activo: El paquete se instalará solo para ese entorno. Si usas `pip install`, y el comando `pip` no está instalado en el entorno que tienes activo, usará por defecto el **`pip` global**, y eso **no** es lo que quieres.

¿Por qué no? ¿Recuerdas el problema con las **dependencias** que mencioné en la sección de entornos virtuales? ¡Pues por eso! El instalador `conda` asume que se usan todos los paquetes que forman parte de su repositorio y realiza un seguimiento de la complicada red de dependencias entre ellos (para obtener más información sobre esto, consulta este enlace[31]).

Para obtener más información sobre las diferencias entre `conda` y `pip`, puedes leer "Understanding Conda and Pip."[32]

Como regla general, primero intenta instalar el paquete con `conda install` y, solo si no lo encuentra ahí, utiliza `pip install`, como hicimos con `torchviz`.

De lo contrario, te recomiendo que aprendas más sobre el tema; **definitivamente** te será útil más adelante. Por el momento, te mostraré los conceptos básicos para que puedas usar `git` para **clonar el repositorio** que contiene todo el código presente en este libro y obtener tu **propia copia local** para modificarlo y experimentar con él como quieras.

En primer lugar, necesitas instalarlo. Así que ve a la página de descargas (*https://git-scm.com/downloads*) y sigue las instrucciones correspondientes a tu sistema operativo. Una vez completada la instalación, abre un **terminal nuevo** o **prompt de Anaconda** (no importa si cierras el anterior). En el nuevo terminal o prompt de Anaconda, deberías poder **ejecutar comandos `git`**.

Para clonar el repositorio de este libro, solo necesitas ejecutar:

```
(pytorchbook)$ git clone https://github.com/dvgodoy/\
PyTorchStepByStep.git
```

El comando anterior creará una carpeta `PyTorchStepByStep` que contiene una copia local de todo el código disponible en el repositorio de GitHub.

7. Jupyter

Después de clonar el repositorio, ve a la carpeta `PyTorchStepByStep` y, **una vez dentro de ella**, **inicia Jupyter** en tu terminal o prompt de Anaconda:

```
(pytorchbook)$ jupyter notebook
```

Esto abrirá tu navegador y verás **la página de inicio de Jupyter** que contiene los notebooks del repositorio y el código correspondiente.

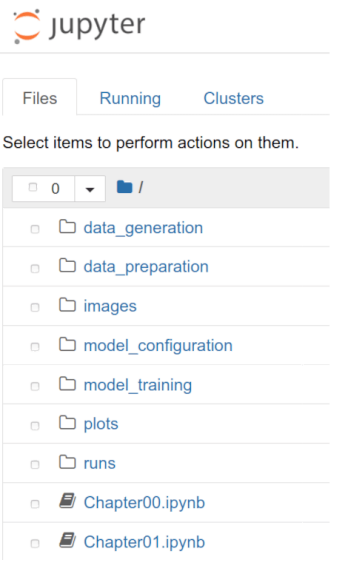

Figura S.4 - Ejecutando Jupyter

Seguimos

Independientemente de cuál de los tres entornos elijas, ahora estás en disposición de continuar y abordar el desarrollo de tu primer modelo PyTorch, **paso a paso**.

[17] https://pytorch.org/docs/stable/notes/randomness.html

[18] https://colab.research.google.com/notebooks/intro.ipynb

[19] https://mybinder.readthedocs.io/en/latest/

[20] https://www.anaconda.com/products/individual

[21] https://bit.ly/2MVk0CM

[22] https://pytorch.org/

[23] https://developer.nvidia.com/cuda-zone

[24] https://www.tensorflow.org/tensorboard

[25] https://www.graphviz.org/

[26] https://bit.ly/3fIwYA5

[27] https://bit.ly/30Ayct3

[28] https://github.com/szagoruyko/pytorchviz

[29] https://docs.anaconda.com/anaconda/packages/pkg-docs/

[30] https://pypi.org/

[31] https://bit.ly/37onBTt

[32] https://bit.ly/2AAh8J5

Capítulo 0
Visualizando el Descenso de Gradiente

Spoilers

En este capítulo:

- definiremos un **modelo de regresión lineal simple**
- recorreremos **cada paso del descenso de gradiente**: inicialización de parámetros, realizar un forward pass, calcular errores y pérdidas, calcular gradientes y actualizar los parámetros
- comprenderemos el concepto de **gradiente** usando **ecuaciones**, **código** y **geometría**
- comprenderemos la diferencia entre **batch**, **mini-batch** y **descenso de gradiente estocástico**
- visualizaremos el impacto que tiene la *tasa* de aprendizaje sobre la **pérdida** (*loss*)
- comprenderemos la importancia de **estandarizar / escalar variables**
- ¡y mucho, mucho más!

No hay código PyTorch en este capítulo... es *Numpy* todo el tiempo porque nuestro objetivo aquí es entender, por dentro y por fuera, cómo funciona el descenso de gradiente. Empezaremos con PyTorch en el siguiente capítulo.

Jupyter Notebook

El Jupyter notebook correspondiente al Capítulo 0[33] es parte del repositorio oficial ***Deep Learning with PyTorch Step-by-Step*** de GitHub. También puedes ejecutarlo directamente en **Google Colab**[34].

Si estás utilizando una *instalación local*, abre un terminal o prompt de Anaconda y ve a la carpeta `PyTorchStepByStep` que clonaste de GitHub. A continuación, *activa* el entorno `pytorchbook` y ejecuta `jupyter notebook`:

```
$ conda activate pytorchbook

(pytorchbook)$ jupyter notebook
```

Si estás utilizando la configuración predeterminada de Jupyter, este enlace debería abrir el notebook del Capítulo 0. Si no es así, simplemente haz clic en `Chapter00.ipynb` en la página de inicio de Jupyter.

Imports

Para organizarnos mejor, al principio de cada capítulo se importarán todas las bibliotecas que se usen en el código utilizado en ese capítulo. Para este capítulo, necesitaremos los siguientes imports:

```
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import StandardScaler
```

Visualizando el Descenso de Gradiente

Según Wikipedia[35]: "***Descenso de Gradiente*** *(Gradient Descent) es un algoritmo de optimización iterativo de primer orden para encontrar un mínimo local de una function diferenciable.*"

Pero yo diría: "El **Descenso de Gradiente** es una técnica iterativa comúnmente utilizada en el aprendizaje automático y el aprendizaje profundo para encontrar el mejor conjunto posible de parámetros / coeficientes para un modelo dado, un conjunto de datos y una función de pérdida, a partir de un estado inicial, generalmente aleatorio."

"*¿Por qué **visualizar** el descenso de gradiente?*"

Creo que *la forma en que generalmente se explica el descenso del gradiente no es muy intuitiva*. A los estudiantes y los principiantes se les presenta un *montón de ecuaciones* y *recetas*: **esa no es la forma en que uno debe aprender un concepto tan importante**.

Si **realmente entiendes** cómo funciona el descenso de gradiente, también entenderás cómo las **características de tus datos** y tu **elección de hiperparámetros** (tamaño de mini-batch y tasa de aprendizaje, por ejemplo) tienen un **impacto** en lo **bueno** y lo **rápido** que va a ser el entrenamiento del modelo.

Cuando digo *entender*, no me refiero a desarrollar manualmente las ecuaciones: esto tampoco desarrolla la intuición. Me refiero a **visualizar** los efectos de diferentes configuraciones; me refiero a **contar una historia** para ilustrar el concepto. Así es como **desarrollas la intuición**.

Dicho esto, cubriré los **cinco pasos básicos** que tendrías que seguir para usar el descenso de gradiente. Te mostraré el código *Numpy* correspondiente mientras te explico muchos **conceptos fundamentales** por el camino.

Pero primero, necesitamos un conjunto de **datos** para trabajar. En lugar de usar cualquier conjunto *externo de datos*,

- definamos qué **modelo** queremos entrenar para comprender mejor el descenso del gradiente; y
- vamos a **generar datos sintéticos** para ese modelo.

Modelo

El modelo debe ser **simple** y **familiar**, para que puedas concentrarte en el **funcionamiento interno** del descenso de gradiente.

Por lo tanto, empezaré con un modelo tan simple como se pueda: una **regresión lineal con una sola variable, *x***.

$$y = b + wx + \epsilon$$

Ecuación 0.1 - Modelo simple de regresión lineal

En este modelo, usamos una **variable (*x*)** para intentar predecir el valor de una **etiqueta (*y*)**. Hay tres elementos en nuestro modelo:

- **parámetro *b***, el *sesgo* (*bias* o *intercepción*), que nos indica el valor medio esperado de *y* cuando *x* es cero
- **parámetro *w***, el *peso* (o *pendiente*), que nos dice cuánto aumenta *y*, en promedio, si aumentamos *x* en una unidad

- y ese **último término** (¿por qué *siempre* tiene que ser una letra griega?), *epsilon*, que está ahí para dar cuenta del **ruido** inherente; es decir, el **error** del que no podemos deshacernos

También podemos concebir la misma estructura del modelo de una manera menos abstracta:

salario = salario mínimo + aumento por año * años de experiencia + ruido

Y para hacerlo aún más concreto, digamos que el **salario mínimo** es de **$1,000** (no es importante qué moneda sea o qué periodo de tiempo consideremos). Con lo cual, si no tienes **ninguna experiencia**, tu salario va a ser el **salario mínimo** (parámetro ***b***).

Además, digamos que, **en promedio**, obtienes un aumento de **$2,000** (parámetro ***w***) por cada año de experiencia que acumules. Por lo tanto, si tienes **dos años de experiencia**, se espera que ganes un salario de **$5,000**. Pero tu salario real es de **$5,600** (¡bien por ti!). Dado que el modelo no puede dar cuenta de esos **$600** extra, tu dinero extra es, técnicamente hablando, **ruido**.

Generación de Datos

Ya conocemos nuestro modelo. Para generar **datos sintéticos**, necesitamos escoger valores para los **parámetros** del modelo. He elegido ***b* = 1** y ***w* = 2** (en miles de dólares) del ejemplo anterior.

Primero, vamos a generar nuestra **variable (*x*)**: Usamos el método `rand()` de *Numpy* para generar aleatoriamente 100 (*N*) puntos entre 0 y 1.

A continuación usamos nuestra **variable (*x*)** y nuestros **parámetros *b* y *w*** en la **ecuación** para calcular nuestras **etiquetas (*y*)**. Pero necesitamos agregar algo de **ruido gaussiano**[36] **(*epsilon*)** también; de lo contrario, nuestro conjunto de datos sintéticos sería una línea perfectamente recta. Podemos generar ruido usando el método `randn()` de_Numpy_, que extrae muestras de una distribución normal (de media 0 y varianza 1), y luego lo multiplicamos por un **factor** para ajustar al **nivel de ruido**. Como no quiero añadir demasiado ruido, he elegido un factor de 0.1.

Generación de Datos Sintéticos

Generación de Datos

```
1 true_b = 1
2 true_w = 2
3 N = 100
4
5 # Generación de Datos
6 np.random.seed(42)
7 x = np.random.rand(N, 1)
8 epsilon = (.1 * np.random.randn(N, 1))
9 y = true_b + true_w * x + epsilon
```

¿Te diste cuenta del método `np.random.seed (42)` en la línea 6? Esta línea de código es en realidad más importante de lo que parece. Garantiza que, cada vez que ejecutamos este código, se generarán los **mismos números aleatorios**.

"*Espera; ¿qué?! ¿No se supone que los números son **aleatorios**? ¿Cómo van a ser los **mismos** números?*" te preguntarás, tal vez incluso un poco molesto por ello.

Números (no tan) Aleatorios

Bueno, ya sabes, los números aleatorios no son en realidad **tan** aleatorios... Son en realidad **pseudoaleatorios**, lo que significa que el generador de números de *Numpy* nos devuelve una **secuencia de números** que **parece aleatoria**. Pero no lo es, en realidad.

Lo **bueno** de este comportamiento es que podemos decirle al generador que **inicie una secuencia particular de números pseudoaleatorios**. Hasta cierto punto, funciona como si le dijéramos al generador: "*por favor, genera la secuencia #42*", y nos da una secuencia de números. Ese número, 42, que funciona como el *índice* de la secuencia, se llama **seed** (*semilla*). Cada vez que le damos la **misma seed**, genera los **mismos números**.

Esto significa que tenemos lo **mejor de ambos mundos**: por un lado, **generamos** una secuencia de números que, a todos los efectos, se **considera aleatoria**; por otro lado, tenemos el **poder de reproducir cualquier secuencia dada**. ¡No te puedes imaginar lo conveniente que puede ser eso para **programas** y encontrar *bugs*!

Además, se garantizara así que **otras personas puedan reproducir tus resultados**. Imaginate lo molesto que sería ejecutar el código presente en este libro y que obtengas diferentes resultados cada vez, teniendo que preguntarte si hay algo mal en él. Pero como ya he establecido una *seed*, tanto tú como yo podemos lograr los mismos resultados, incluso si se trata de generar datos aleatorios!

A continuación, vamos a **dividir** nuestro conjunto de datos sintéticos en dos, uno para **entrenar** (*train*) y otro para **validar** (*validation*), desordenando al azar las filas (cada una con su índice) y utilizando las primeras 80 filas para el entrenamiento.

"*¿Por qué necesitas **desordenar** los datos generados aleatoriamente? ¿No son lo suficientemente aleatorios?*"

Sí, lo **son**, y desordenarlos es redundante en este ejemplo. Pero obtendrás **mejores resultados** si desordenas al azar tus datos antes de entrenar un modelo, ya que eso hará que el descenso de gradiente sea más efectivo.

Hay una **excepción** a la regla de "siempre desordenar aleatoriamente": los problemas de **series temporales**, donde desordenar aleatoriamente puede dar lugar a la fuga de información entre el conjunto de datos para entrenar el modelo y el conjunto que dejas para validarlo.

Partición Entrenamiento-Validación-Test

El objetivo de este libro no es explicar el razonamiento detrás de la **partición Entrenamiento-Validación-Test** del conjunto de datos, pero hay dos cosas que me gustaría explicar:

1. La partición debe **siempre** ser la **primera cosa** que hagas: antes de cualquier procesamiento que necesites hacer a los datos, o transformaciones o lo que

sea; **nada debe afectar los datos antes de la partición**. Por eso es que hacemos esto **inmediatamente después de la generación de los datos sintéticos**.

2. En este capítulo usaremos **solo el conjunto de entrenamiento**, por lo que no me molesté en crear un **conjunto para el test**, pero he hecho la partición de todos modos para **hacer hincapié en el punto #1** :-)

Partición Entrenamiento-Validación

```
# Desordena los índices
idx = np.arange(N)
np.random.shuffle(idx)
# Los primeros 80 índices aleatorios para el entrenamiento
train_idx = idx[:int(N*.8)]
# Los índices restantes para la validación
val_idx = idx[int(N*.8):]
# Genera los conjuntos de datos para entrenamiento y validación
x_train, y_train = x[train_idx], y[train_idx]
x_val, y_val = x[val_idx], y[val_idx]
```

"*¿Por qué no has usado* `train_test_split()` *directamente de Scikit-Learn?*" te preguntarás.

Ese es un buen punto. Más adelante, nos referiremos a los **índices de los datos** pertenecientes a conjuntos de entrenamiento o validación, en lugar de los datos en sí. Así que pensé que sería conveniente usarlos desde el principio.

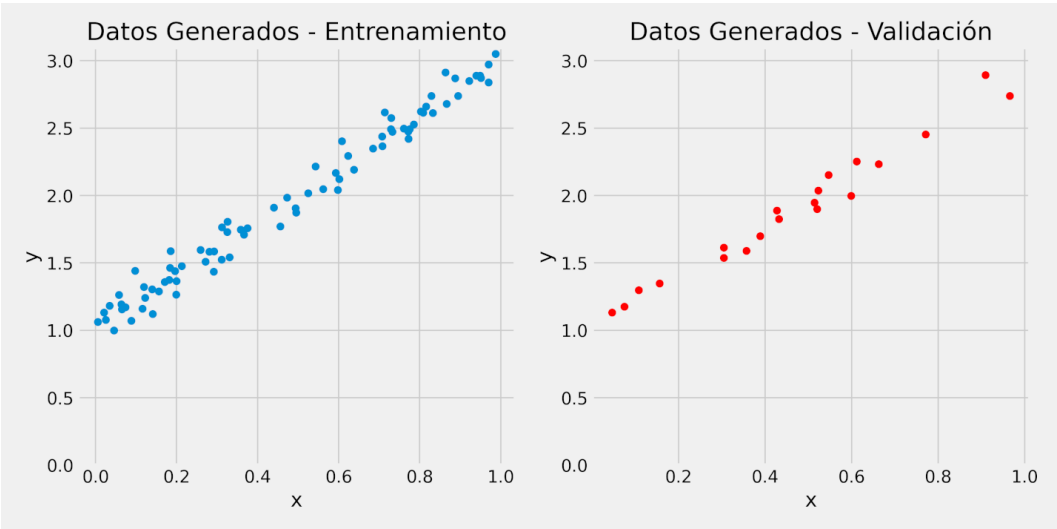

Figura 0.1 - Datos sintéticos: conjuntos de entrenamiento y validación

Sabemos que ***b* = 1**, ***w* = 2**, pero ahora veamos cuánto nos podemos **aproximar** a los valores verdaderos usando **descenso de gradiente** y los 80 puntos en el **conjunto de entrenamiento** (***N* = 80**).

Paso 0 - Inicialización aleatoria

En nuestro ejemplo, ya **conocemos** los **valores verdaderos** de los **parámetros**, pero esto obviamente nunca sucederá en la vida real: Si *conociéramos* los valores verdaderos, ¿por qué molestarse en entrenar un modelo para encontrarlos?

Vale, dado que **nunca sabremos** el **valor verdadero** de los parámetros, necesitamos darles un **valor inicial*. ¿Cómo los elegimos? Resulta que un** valor aleatorio** es tan bueno como cualquier otro.

Aunque la inicialización es **aleatoria**, hay ciertos **esquemas de inicialización** inteligentes que deben usarse cuando entrenemos modelos más sofisticados. Volveremos a ellos (mucho) más adelante, en el segundo volumen de la serie.

Para entrenar un modelo, debes **inicializar aleatoriamente los parámetros / pesos** (Tenemos sólo dos, ***b*** y ***w***).

Inicialización Aleatoria

```
1 # Paso 0 - Inicializa los parámetros "b" y "w" aleatoriamente
2 np.random.seed(42)
3 b = np.random.randn(1)
4 w = np.random.randn(1)
5 print(b, w)
```

Output

```
[0.49671415] [-0.1382643]
```

Paso 1 - Calcular las Predicciones del Modelo

Este es el **pase hacia adelante** (*forward pass*); simplemente *calcula las predicciones del modelo utilizando los valores actuales de los parámetros / pesos*. Al principio,

produciremos **predicciones realmente malas**, ya que comenzamos con **valores aleatorios en el Paso 0**.

Paso 1

```
# Paso 1 - Calcula las predicciones de nuestro modelo
# forward pass
yhat = b + w * x_train
```

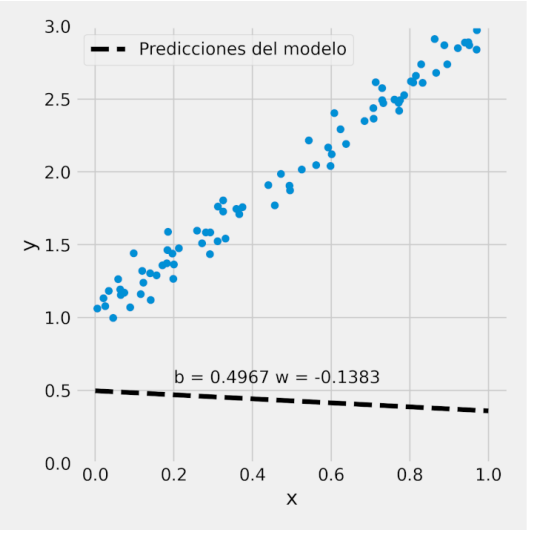

Figura 0.2 - Predicciones del modelo (con parámetros aleatorios)

Paso 2 - Calcular la pérdida (*Loss*)

Hay una diferencia sutil pero fundamental entre **error** y **pérdida**.

El **error** es la diferencia entre el **valor real (etiqueta)** y el **valor predicho** calculado para un único dato. Así que, para un punto *i*-ésimo dado (de nuestro conjunto de datos de *N* puntos), su error es:

$$\text{error}_i = \hat{y}_i - y_i$$

Ecuación 0.2 - Error

El error del **primer punto** de nuestro conjunto de datos ($i = 0$) puede ser representado así:

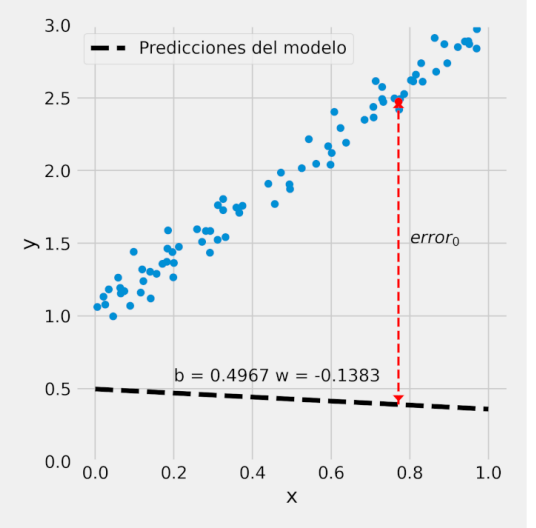

Figura 0.3 - Error de predicción (para un solo dato)

La **pérdida**, por otro lado, es una especie de **agregación de errores para un conjunto de datos**.

Parece bastante evidente cómo calcular la pérdida correspondiente a **todos** (*N*) los puntos, ¿verdad? Bueno, si y no. Aunque seguramente se llegará de **manera más eficiente** desde los **parámetros aleatorios** iniciales hasta los parámetros que **minimizan la pérdida**, también será seguramente demasiado **lento**.

Esto significa que seguramente *tenemos que sacrificar (un poco) de estabilidad para a cambio ir más rápidos en la optimización*. Esto se logra fácilmente eligiendo al azar (*sin reemplazo*) un subconjunto de ***n*** de los ***N*** datos cada vez que calculamos la pérdida.

Batch, **Mini-batch**, *y* **Descenso de Gradiente Stocástico**

- Si utilizamos **todos los puntos** en el conjunto de entrenamiento (*n* = ***N***) para calcular la pérdida, estamos realizando un gradiente de descenso por **batches**;
- Si usáramos un **solo punto** (*n* = **1**) cada vez, sería un descenso de gradiente **estocástico**;
- Cualquier otra cosa (*n*) **entre 1 y *N*** correspondería a un descenso de gradiente por **mini-batches**;

Para un problema de regresión, la **pérdida** viene dada por el **error cuadrático medio (MSE, del inglés *mean squared error*)**; es decir, la media de todas las

diferencias al cuadrado entre **etiquetas** (*y*) y **predicciones** (*b* + *wx*).

$$\begin{aligned}\text{MSE} &= \frac{1}{n}\sum_{i=1}^{n} \text{error}_i{}^2 \\ &= \frac{1}{n}\sum_{i=1}^{n} (\hat{y}_i - y_i)^2 \\ &= \frac{1}{n}\sum_{i=1}^{n} (b + wx_i - y_i)^2\end{aligned}$$

Ecuación 0.3 - Pérdida: error cuadrático medio (MSE)

En el siguiente código, usamos **todos los datos** del conjunto de entrenamiento para calcular la **pérdida**, por lo que ***n* = *N* = 80**, lo que significa que estamos realizando un **descenso de gradiente por batches**.

Step 2

```
# Step 2 - Calculando la pérdida
# Usamos TODOS los datos, así que esto es descenso de gradiente
# por BATCHES.
# ¿Cuánto se equivoca nuestro modelo? ¡Ese es el error!
error = (yhat - y_train)

# Es una regresión, por lo que se calcula el error
# cuadrático medio (MSE)
loss = (error ** 2).mean()

print(loss)
```

Output

```
2.7421577700550976
```

Superficie de Pérdidas

Acabamos de calcular la **pérdida** (2.74) correspondiente a nuestros **parámetros inicializados aleatoriamente** (*b* = 0.49 y *w* = -0.13). ¿Qué pasaría si hiciéramos lo mismo para **TODOS** los valores posibles de ***b*** y ***w***? Bueno, no *todos* los valores

posibles, sino *todas las combinaciones de valores uniformemente espaciados en un rango determinado*, como:

```
# Recordatorio:
# true_b = 1
# true_w = 2

# tenemos que dividir los rangos en 100 intervalos uniformemente
# espaciados
b_range = np.linspace(true_b - 3, true_b + 3, 101)
w_range = np.linspace(true_w - 3, true_w + 3, 101)
# meshgrid es una función muy útil para generar una matriz (grid)
# de valores de b y w para todas las combinaciones
bs, ws = np.meshgrid(b_range, w_range)
bs.shape, ws.shape
```

Output

```
((101, 101), (101, 101))
```

El resultado de la operación `meshgrid()` fueron dos matrices (101, 101) que representan los valores de cada parámetro dentro de una cuadrícula. ¿Qué pinta tiene una de estas matrices?

```
bs
```

Output

```
array([[-2.  , -1.94, -1.88, ...,  3.88,  3.94,  4.  ],
       [-2.  , -1.94, -1.88, ...,  3.88,  3.94,  4.  ],
       [-2.  , -1.94, -1.88, ...,  3.88,  3.94,  4.  ],
       ...,
       [-2.  , -1.94, -1.88, ...,  3.88,  3.94,  4.  ],
       [-2.  , -1.94, -1.88, ...,  3.88,  3.94,  4.  ],
       [-2.  , -1.94, -1.88, ...,  3.88,  3.94,  4.  ]])
```

Es cierto que estamos haciendo un poco de *trampa* aquí, ya que sabemos los **verdaderos** valores de *b* y *w*, con lo cual podemos elegir los **rangos perfectos** para los parámetros. Pero es sólo con fines educativos :-)

A continuación, podríamos usar esos valores para calcular las **predicciones**, **errores** y **pérdidas** correspondientes. Comencemos tomando un **único dato** del conjunto de entrenamiento y calculando las predicciones para cada combinación en nuestra cuadrícula:

```
dummy_x = x_train[0]
dummy_yhat = bs + ws * dummy_x
dummy_yhat.shape
```

Output

```
(101, 101)
```

Gracias a la naturaleza vectorial de *Numpy*, esta biblioteca es capaz de entender que queremos multiplicar el **mismo valor *x*** por **cada entrada** en la **matriz *ws***. Esta operación dio como resultado una **cuadrícula de predicciones** para ese **único dato**. Ahora tenemos que repetir esto para **cada uno de nuestros 80 datos** del conjunto de entrenamiento.

Podemos usar el método `apply_along_axis()` de *Numpy* para lograr esto:

Mira mamá, ¡sin bucles!

```
all_predictions = np.apply_along_axis(
    func1d=lambda x: bs + ws * x,
    axis=1,
    arr=x_train,
)
all_predictions.shape
```

Output

```
(80, 101, 101)
```

¡Perfecto! Tenemos **80 matrices** de tamaño (101, 101), **una matriz para cada dato**, cada matriz contiene una **cuadrícula de predicciones**.

Los **errores** son la diferencia entre las predicciones y las etiquetas, pero no

podemos realizar esta operación de inmediato; necesitamos trabajar un poco en nuestras **etiquetas (*y*)**, para que tengan el **tamaño** adecuado para ello (la vectorización es buena, pero no *tan* buena):

```
all_labels = y_train.reshape(-1, 1, 1)
all_labels.shape
```

Output

```
(80, 1, 1)
```

Nuestras **etiquetas** resultaron ser **80 matrices de tamaño (1, 1)** —el tipo de matriz más aburrida— pero es suficiente para que la vectorización haga su magia. Ahora ya podemos calcular los **errores**:

```
all_errors = (all_predictions - all_labels)
all_errors.shape
```

Output

```
(80, 101, 101)
```

Cada predicción tiene su propio error, por lo que obtenemos **80 matrices** de tamaño (101, 101), una vez más, una matriz para cada dato, donde cada matriz contiene una **cuadrícula de errores**.

El único paso que falta es calcular el **error cuadrático medio**. Primero, tomamos el cuadrado de todos los errores. Luego, **promediamos los cuadrados sobre todos los datos**. Dado que nuestros datos están en la **primera dimensión**, usamos `axis=0` para calcular esta media:

```
all_losses = (all_errors ** 2).mean(axis=0)
all_losses.shape
```

Output

```
(101, 101)
```

El resultado es una **cuadrícula de pérdidas**, una matriz de tamaño (101, 101), **cada pérdida** correspondiente a una **combinación diferente de los parámetros *b* y *w***.

Esto es lo que llamamos la **superficie de pérdidas** (*loss surface*), que se puede visualizar en una gráfica 3D, donde el eje vertical (*z*) representa los valores de la pérdida. Si **conectamos** las combinaciones de *b* y *w* que dan lugar al mismo **valor de pérdida**, obtendremos una **elipse**. Así que podemos dibujar esta elipse en el plano original *b* x *w* (en azul, para un valor de pérdida de 3). Esto es, de hecho, lo que un **diagrama de contornos** hace. A partir de ahora, siempre usaremos el diagrama de contornos, en lugar de la versión 3D correspondiente.

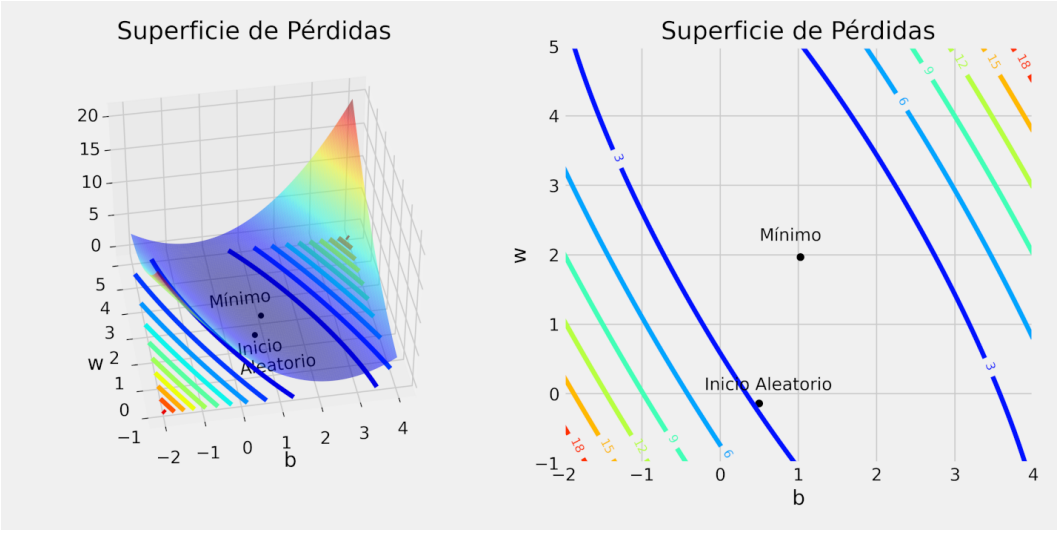

Figura 0.4 - Superficie de pérdidas

En el centro de la gráfica, donde los parámetros (*b*, *w*) tienen valores cercanos a (1, 2), la pérdida está en su valor **mínimo**. Este es el punto que estamos tratando de alcanzar usando el descenso de gradiente.

Abajo, un poco a la izquierda, está el punto de **comienzo aleatorio**, correspondiente a nuestros valores inicializados aleatoriamente.

Esta es una de las cosas buenas de abordar un problema simple como una regresión lineal con una sola variable: Tenemos solo **dos parámetros**, con lo cual **podemos calcular y visualizar la superficie de pérdidas**.

Desgraciadamente, para la gran mayoría de los problemas, **calcular la superficie de pérdidas no va a ser factible**: tenemos que confiar en la capacidad del descenso de gradiente para alcanzar un punto de valor mínimo, incluso si comienza en algún punto aleatorio.

Puedes Leer Más

Comprar la versión completa: https://leanpub.com/pytorch_ES