Shantal Chavarría Siles

Deep Learning with Python

# What is Deep learning

Para aclarar primero que es Deep Learning, debemos aclarar antes qué es inteligencia artificial y machine learning. ¿Cómo estos se relacionan entre si?

Diagrama, Diagrama de Venn

Descripción generada automáticamente

## Inteligencia Artificial

### Podría decirse que la IA nació debido a “el esfuerzo por automatizar las tareas intelectuales normalmente realizadas por humanos” (Chollet, 2018)

## Machine Learning

### Un sistema de aprendizaje automático (machine learning) está entrenado en lugar de programado explícitamente. En lugar de introducir reglas y datos para ser procesados de acuerdo con esas reglas, para obtener respuestas. Con machine Learning introducimos datos y las respuestas esperadas de esos datos, para obtener reglas.

### Diagrama Descripción generada automáticamente

### Machine Learning cuando le son dados ejemplos de lo que se espera obtener, este descubre reglas para ejecutar una tarea de procesamientos de datos.

### Se necesitan tres cosas para hacer ML:

### *Puntos de datos de entrada*: Por ejemplo, si la tarea es reconocimiento de voz, estos puntos de datos serían archivos de sonidos de personas hablando.

### *Ejemplos de los resultados esperados*: Siguiendo el ejemplo de reconocimiento de voz, estos podrían ser transcripciones generadas por humanos de archivos de sonido.

### *Una forma de medir si el algoritmo está haciendo un buen trabajo*: Esta medida es una señal realimentación para ajustar la forma en la que el algoritmo funciona. Este paso de ajuste es lo que llamamos **learning.**

### Es decir, un modelo de ML expone lo que ha aprendido mostrando ejemplos reconocidos de entradas y salidas.

## El “Deep” en Deep Learning

### El aprendizaje profundo es un subcampo específico del aprendizaje automático: una nueva visión del aprendizaje de representaciones a partir de datos que pone énfasis en el aprendizaje de capas sucesivas de representaciones cada vez más significativas. No se refiere a un entendimiento más profundo logrado por el enfoque. A cuantas capas contribuyen a un modelo de datos se denomina *profundidad* del modelo.

### En Deep learning, estas representaciones en capas se aprenden (casi siempre) a través de modelos llamados redes neuronales, estructurados en capas literales apiladas una encima de la otra.

### ¿Cómo son las representaciones aprendidas por un algoritmo de aprendizaje profundo?

### Gráfico Descripción generada automáticamente

### Se puede pensar en una red profunda como una operación de destilación de información de múltiples etapas, donde la información pasa por filtros sucesivos y sale cada vez más depurada (es decir, útil con respecto a alguna tarea)

## Como funciona el Deep Learning en tres figuras.

## Sabemos que las redes neuronales profundas realizan este mapeo de entrada a destino a través de una secuencia profunda de transformaciones de datos simples (capas) y que estas transformaciones de datos se aprenden mediante la exposición a ejemplos. Ahora veamos cómo ocurre este aprendizaje, concretamente.

## En específico lo que una capa hace a sus datos de entrada se almacena en los *pesos* de la capa, que en esencia son un montón de números. Técnicamente se podría decir que la transformación implementada por una capa es *parametrizada* por sus *pesos.* En este contexto, *learning* significa encontrar un conjunto de valores para todas las capas en una red, tales que la red mapee correctamente ejemplos de entradas a sus objetivos asociados. Lo que pasa, es que una red neuronal puede contener 10 millones de parámetros. Encontrar el valor correcto para todos puede ser una tarea desalentadora, en especial por que modificar el parámetro de una puede afectar el comportamiento de las otras.

## Diagrama Descripción generada automáticamente

## Para controlar la salida de la red neuronal, necesitamos ser capaces de medir que tan distante está la salida de lo que esperamos. Este es el trabajo de la *función de pérdida*, también llamada *función objetiva.* Esta función tomas las predicciones de la red y el objetivo real y calcula una puntuación de distancia, capturando qué tan bien ha funcionado la red en este ejemplo específico.

## Diagrama Descripción generada automáticamente

El truco fundamental en Deep Learning es usar esta nota de puntuación como señal de retroalimentación para ajustar el valor de los pesos un poco, en una dirección que bajará la puntuación de pérdida para el ejemplo actual. Este ajuste es el trabajo del *optimizador*, que implementa lo que se llama el algoritmo de *Retropropagación*: el algoritmo central en el aprendizaje profundo.

Diagrama

Descripción generada automáticamente

Inicialmente los pesos de la red se asignan de la manera aleatoria, por lo que la red hace implementaciones a una serie de transformaciones aleatorias. Naturalmente, las salidas son bastante lejanas a las esperadas, y el puntaje de pérdida será muy alto. Pero con cada ejemplo que la red procese, los pesos serán ajustados un poco en la dirección correcta, y el puntaje de pérdida disminuirá. Esto es *ciclo de entrenamiento*.

# Componentes matemáticos de las redes neuronales

## Representación de Datos para Redes Neuronales.

## Los datos se almacenan en arreglos Numpy multidimensionales, también conocidos como *tensores.* Pero ¿qué es un tensor?

## En esencia, un tensor es un contenedor de datos, casi siempre numérica. Las matrices son tensores en 2D; los tensores son una generalización de las matrices a un número arbitrario de dimensiones (en tensores a estas se les llama ejes). Al número de ejes de un tensor también se le llama *rango.*

## Escalares (tensores 0D)

## Un tensos que contenga solo un número es llamado *escalar* (o tensor escalar, o tensores 0D). En Numpy, un número `float32` o `float64` es un tensor escalar (o arreglo escalar). Podemos mostrar el número de ejes de un tensor Numpy mediante el atributo `ndim`; un tensor escalar tiene 0 ejes.

## Aquí hay un escalar Numpy:

## Patrón de fondo Descripción generada automáticamente con confianza baja

## Vectores (tensores 1D)

## Un arreglo de números se llama vector, o tensor 1D. Un tensor 1D tiene exactamente un eje.

## A continuación, un vector Numpy:

## Imagen que contiene Patrón de fondo Descripción generada automáticamente

## Este vector tiene cinco entradas, por eso se llama *vector de 5 dimensiones****.* No hay que confundir un vector 5D con un tensor 5D.** Un vector 5D tiene solo un eje y cinco dimensiones a lo largo de su eje, mientras que un vector 5D tiene 5 ejes y cualquier número de dimensiones a lo largo de cada eje. *Dimensionalidad* puede denotar el número de entradas de un eje (para el caso de un vector) o el número de ejes (para el caso de un tensor), lo cual puede ser confuso a veces. Para evitar confusiones hablaremos de *tensores de rango n,* siendo *n* el número de ejes.

## Matrices (tensores 2D)

## Un arreglo de vectores es una *matriz,* o tensor 2D. Una matriz tiene dos ejes (filas y columnas). Podemos visualizarla como una cuadrícula de rectangular de números.

## Esto es una matriz Numpy:

## Texto Descripción generada automáticamente con confianza baja

## Las entradas del primer eje son las filas, y las entradas del segundo eje son las columnas. En este ejemplo [5, 78, 2, 34, 0] es la primera fila de x, y [5, 6, 7] la primera columna.

## Tensores 3D y tensores de más dimensiones

## Si empaquetamos dichas matrices en una nueva matriz, obtenemos un tensor 3D, que se puede interpretar visualmente como un cubo de números. A continuación, un tensor 3D de Numpy:

## Tabla Descripción generada automáticamente

Empaquetando tensores 3D en una matriz, creamos tensores 4D, y así sucesivamente. En Deep Learning, generalmente manipularemos tensores desde 0D a 4D, aunque podríamos subir a 5D si se procesan datos de video.

## Atributos clave

## Un tensor es definido por tres atributos clave:

### *Número de ejes (rango):* Por ejemplo, un tensor 3D tiene tres ejes, y una matriz tiene dos ejes. Esto también se le conoce como la ndim del tensor en librerías de Python como Numpy.

### *Forma*: Esto es una tupla de enteros que describe cuantas dimensiones tiene el tensor a lo largo de cada eje. Por ejemplo, la matriz anterior tenía forma (3,5), y el tensor 3D tenía forma (3,3,5). Además, un vector tiene forma con un solo elemento, como (5,) y un escalar tiene forma vacía, ().

### *Tipo de Dato* (dtype en librerías de Python): Está es el tipo de dato contenido en el tensor, por ejemplo, un tipo de tensor puede ser float32, uint8, float64, etc. En raras ocasiones podría ser tipo char, aunque este tipo de tensores no existe en Numpy.

### Para observar mejor esto, cargaremos los datos de MNIST del paquete keras:

### 

### Ahora desplegaremos el número de ejes del tensor train\_images, con el atributo ndim:

### 

### Ahora su forma con shape:

### 

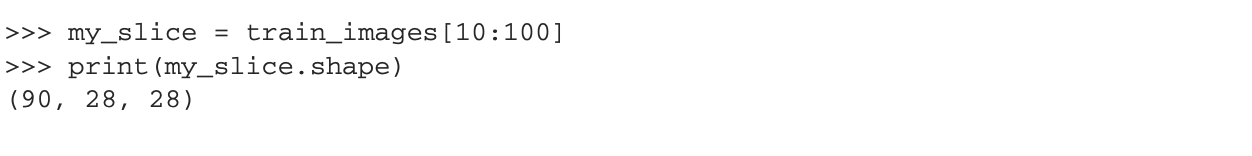
### Y su tipo de datos, con el atributo dtype:

### 

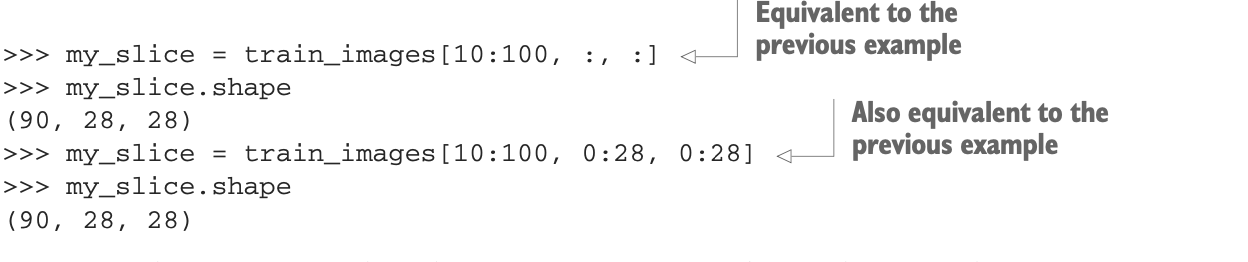
## Manipulando tensores en Numpy

En el ejemplo anterior, seleccionamos un digito específico del primer eje usando la sintaxis train\_images [i], a esto se le conoce *corte de tensor*. Veamos más operaciones de corte de tensor que puede realizar en matrices Numpy.

El siguiente ejemplo selecciona los dígitos del #10 al #100 (100 no está incluido) y los pone en un arreglo de forma (90,28,28):



Esto es equivalente a esta notación más detallada, la cual especifica un índice de inicio y un índice de fin para el corte junto con cada eje. Note que “:” es equivalente a seleccionar todo el eje entero:



## La noción de lotes de datos

Los modelos de Deep Learning no procesan un conjunto de datos de una vez; más bien, dividen los datos en lotes más pequeños.

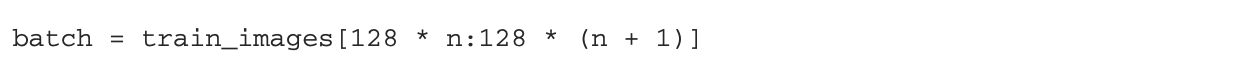
Aquí hay un lote de tamaño 128 de nuestra tabla MNIST:



El siguiente lote sería:



Y el n-ésimo:



Cuando se habla de tensor de lote, al primer eje (eje 0) se le llama *batch axis* o *batch dimension.* Esto es un término utilizado frecuentemente cuando se usa *Keras* y otras librerías de Deep Learning.

## Ejemplos de tensores en la vida real.

Los tensores de datos que se manipularán casi siempre caerán en una de las siguientes categorías:

* Datos Vectoriales (Vector Data): Tensores 2D
* Series de tiempo: Tensores 3D
* Imágenes: Tensores 4D
* Video: Tensores 5D

# oPeraciones con Tensores

Todas las transformaciones aprendidas por Redes Neuronales se pueden reducir a un puñado **de operaciones de tensores** aplicadas a tensores de datos numéricos. Por ejemplo, es posible agregar tensores, multiplicar tensores, etc.

## Operaciones por entradas

La operación *relu* y la adición son operaciones que se aplican independientemente entrada por entrada (*relu(x) = max(x,0))*.

Si quisiéramos definir una función de Python de una operación basada en entradas, usaríamos un ciclo *for*, como en esta implementación ingenua de una operación aditiva basada en elementos:

Diagrama

Descripción generada automáticamente

Pero con Numpy podemos hacer operaciones entrada por entrada de una forma mucho más rápida:

Texto

Descripción generada automáticamente

## “Broadcasting”

## El ejemplo anterior *naive\_add* solo recibe tensores 2D con dimensiones iguales, pero ¿qué pasa con la adición cuando las formas de los dos tensores son diferentes entre sí?

## Si esto pasara entonces el tensor más pequeño sería “*broadcasted”* para que coincida con la forma del tensor más grande, esto se logra en dos pasos:

###### Ejes (llamados ejes de difusión) se agregan al tensor más pequeño para que coincida con el *ndim* del tensor más grande.

###### El tensor más pequeño se repite junto a estos nuevos ejes para que coincida con la forma del tensor más grande.

Veamos un ejemplo concreto. Considere X con forma (32, 10) y Y con forma (10,). Primero, agregamos un primer eje vacío a Y, su nueva forma sería (1, 10). Luego, repetimos y 32 veces a lo largo de este nuevo eje, de modo que terminamos con un tensor Y1 con forma (32, 10), donde Y1 [i,j] == Y para i en el rango (0, 32). En este punto, podemos sumar X y Y1, porque tienen la misma forma.

## Punto Tensor

## La operación punto, o producto tensor, es la operación más útil y común en tensores. Esta combina operaciones en las entradas de los tensores introducidos.

## Los productos entrada por entrada se hacen con el operador “\*” en Numpy, Keras, Theano y TensorFlow. *dot* por su parte utiliza la misma sintaxis en Numpy y Keras, pero una diferente en TensorFlow:

## 

## Pero ¿qué hace matemáticamente esta operación? Primero veamos el operador *dot* de dos vectores *x* y *y:*

## Gráfico Descripción generada automáticamente con confianza baja

## Note que, el producto punto entre dos vectores es un escalar y que solo vectores con las mismas dimensiones son compatibles para realizar producto punto.

## El producto punto también se puede realizar en matrices. El producto punto de una matriz X y una matriz Y, nos retornaría un vector cuyos coeficientes son el producto punto entre las filas de X y Y. Sin embargo, esto sucede siempre y cuando el número de filas de X coincida con el número de columnas de Y, esto nos dará como resultado una matriz de forma *(x.shape[0],y.shape[1]):*

## Diagrama, Esquemático Descripción generada automáticamente

## El producto punto también se puede generalizar a tensores con un número arbitrario de ejes, aunque es más común en matrices.

## Para entender la compatibilidad de las dimensiones en el producto punto se muestra la siguiente figura:

## Diagrama Descripción generada automáticamente

## Interpretación geométrica de operaciones entre tensores

Todas las operaciones de tensor tienen una interpretación geométrica, debido a que estas se pueden ver como coordenadas en un espacio geométrico. Por ejemplo, consideremos la suma. Comenzaremos con el siguiente vector:



Esto es un punto en un espacio bidimensional. Podemos imaginar un vector como una flecha uniendo el origen al punto.

Imagen en blanco y negro

Descripción generada automáticamente con confianza baja

Ahora consideremos un nuevo punto B, el cual sumaremos a A. Esto geométricamente se hace encadenando las dos flechas de vectores, la nueva ubicación resultante será la representación vectorial de la suma de ambos vectores.

Diagrama

Descripción generada automáticamente

# Optimización basada en gradientes

Los pesos de una capa de una red neuronal contienen información aprendida por la red durante el entrenamiento.

Inicialmente estas matrices de pesos son llenadas con pequeños valores aleatorios (inicialización aleatoria). Aunque los resultados no tengan mucho sentido, sí representan un punto de partida, ya que con base a esto iremos ajustando los pesos, basándonos en la retroalimentación antes mencionada. Este ajuste es lo que se conoce como entrenamiento, es decir, el aprendizaje del machine learning

Todo esto sucede dentro de un ciclo de entrenamiento (training loop), repetiremos estos pasos tanto como sea necesario:

1. Divida los datos en muestras de entrenamiento *X* y prueba *y*.
2. Ejecute la red en *X* (un paso llamado pase directo) para obtener predicciones *y\_pred*.
3. Calcule la pérdida de la red en el lote, una medida de error entre *y\_pred* y *y*.
4. Actualice todos los pesos de la red de manera que reduzca ligeramente el error en este lote.

Eventualmente acabaremos teniendo una red con un error muy bajo. La red habrá “aprendido” a mapear sus entradas a los objetivos buscados. Aquí el verdadero reto consistirá en encontrar los pesos adecuados para la red. ¿Cómo saber si los pesos deben aumentan o disminuir?

Un buen es aprovechar el hecho de que todas las operaciones utilizadas en la red son diferenciables y calcular el gradiente de la pérdida con respecto a los coeficientes de la red. Luego puede mover los coeficientes en la dirección opuesta al gradiente, disminuyendo así la pérdida.

## ¿Qué es una derivada?

Considere una función *continua y suave* f(x) = y; en R. Como la función es *continua*, un pequeño cambio en *x* genera un pequeño cambio en *y*.



Como la función es *suave*, es decir, no tiene pendientes muy pronunciadas, cuando **epsilon\_x** es lo suficientemente pequeño, alrededor de un cierto punto **p**, es posible aproximar *f* como una función lineal de pendiente **a**, de modo que **epsilon\_y** se convierte en **a\* épsilon\_x**:



Cabe resaltar, que esta aproximación es válida solo cuando *x* está lo suficientemente cerca de *p.*

Está pendiente es la *derivada* de *f* en *p.* Si *a* es negativa, esto quiere decir que un pequeño cambio en *x* alrededor de *p* resultaría en una disminución de *f(x).* Si por otra parte, *a* fuera positiva, un pequeño cambio en *x* resultaría en un aumento de *f(x).* Además, el valor absoluto de *a* (la magnitud de la derivada) indicará qué tan rápido ocurrirá este aumento o disminución.

Imagen que contiene Texto

Descripción generada automáticamente

Para cada función diferenciable *f(x)* (que se puede derivar), existe una función derivada *f ’(x)* que mapea valores de x a la pendiente de la aproximación lineal local de *f* en esos puntos.

Por ejemplo, la derivada de *cos(x)* es *-sen(x),* la derivada de *f(x) = a\*x* es *f ‘ (x) = a*, etc.

En conclusión, la derivada describe como *f(x)* evoluciona al cambiar *x.* Si deseáramos minimizar *f(x),* solo debemos mover *x* en la dirección opuesta a la derivada

## Derivada de una operación tensorial: El Gradiente

El *gradiente* es la derivada de una operación tensorial. Es la generalización del concepto de derivadas a funciones que toman tensores como entradas.

Considere un vector de entrada *x*, una matriz *W*, un objetivo *y* y una función de pérdida *loss*. Podemos utilizar W para calcular un *y\_pred* y calcular la pérdida o la diferencia entre el candidato objetivo *y\_pred* y el objetivo *y*:



Digamos que el valor actual de W es W0. Entonces la derivada de f en el punto W0 es un gradiente tensorial (f) (W0) con la misma forma que W, donde cada gradiente de coeficiente (f) (W0) [i, j] indica la dirección y magnitud del cambio en *loss\_value* que se observa al modificar W0 [i, j]. Ese gradiente tensorial (f) (W0) es el gradiente de la función f (W) = *loss\_value* en W0.

Anteriormente viste que la derivada de una función *f (x)* de un coeficiente simple se puede interpretar como la pendiente de la curva de f. Asimismo, el gradiente *(f) (W0)* se puede interpretar como el tensor que describe la curvatura de *f (W)* alrededor de W0.

Por esta razón, de la misma manera que, para una función f (x), puede reducir el valor de f (x) moviendo x un poco en la dirección opuesta a la derivada, con una función f (W) de a tensor, puede reducir f (W) moviendo W en la dirección opuesta al gradiente.

Por ejemplo, W1 = W0 - *paso* \* gradiente (f) (W0) (donde *paso* es un factor de escala pequeño). Eso significa ir en contra de la curvatura, que intuitivamente debería ubicarte más abajo en la curva. Tenga en cuenta que el paso del factor de escala es necesario porque el gradiente (f) (W0) solo se aproxima a la curvatura cuando está cerca de W0, por lo que no quiere alejarse demasiado de W0.

Interpretando el gradiente en dos dimensiones

Supongamos que f está en un plano bidimensional, su gradiente contiene toda la información de sus derivadas parciales en un vector.

Es decir, f es una función vectorial. Ahora sea ( un punto en el espacio. ¿Qué nos dice El gradiente entonces apuntará en la dirección para incrementar f, en dirección a la cima de la función

Posibles Ejercicios

Funciones Naive de Operaciones entre tensores