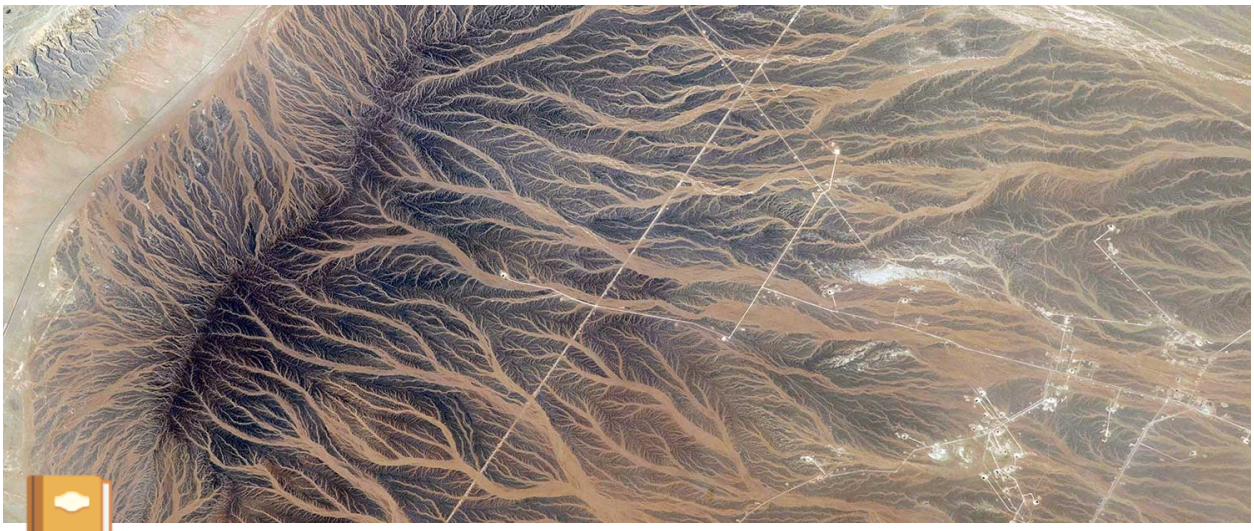


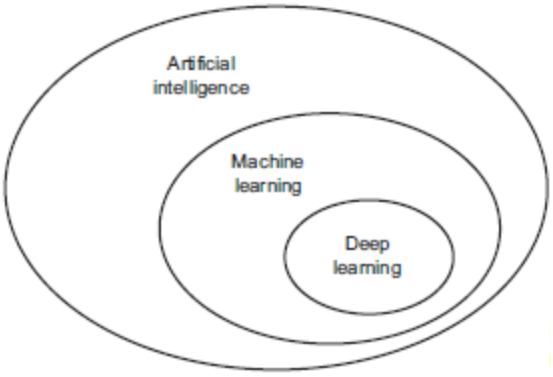
Redes Neuronales Artificiales

Introducción

Las **redes neuronales** se utilizan para generar predicciones, análisis de texto, voz e imágenes y detección de objetos. En esta unidad veremos los conceptos básicos de las redes neuronales, cómo funcionan y la matemática que se encuentre detrás.

Las redes neuronales (Neural Networks) se enmarcan dentro del campo de la Inteligencia Artificial. Nacieron en los años 60, pero no fueron utilizadas hasta décadas más adelante debido a la baja capacidad de cómputo y almacenamiento de la época. Sin embargo, con el crecimiento de las computadoras el interés por ellas aumentó exponencialmente.

Las redes neuronales se relacionan con el **aprendizaje profundo** (*deep learning*) ya que este último se basa en el entrenamiento y composición de la arquitectura de una red neuronal.



El **aprendizaje profundo** es un subcampo específico de aprendizaje automático: una nueva forma de la representación de aprendizaje de los datos que pone énfasis en el aprendizaje de capas sucesivas de representaciones cada vez más significativas.

> Lo *profundo* en el aprendizaje profundo representa la idea de capas sucesivas de representaciones.

La cantidad de capas que componen un modelo de datos se denomina **profundidad** del modelo.

El aprendizaje profundo moderno a menudo implica decenas o incluso cientos de capas sucesivas de representaciones y todos se aprenden automáticamente de la exposición a los datos de entrenamiento. Mientras que, otros enfoques tienden a centrarse en aprender solo una o dos capas de representaciones de los datos. Por lo tanto, a veces se les llama **aprendizaje superficial (*shallow learning*)**.
El término red neuronal es una referencia a la neurobiología, pero aunque algunos de los conceptos centrales en el aprendizaje profundo se desarrollaron en parte al inspirarse en nuestra comprensión del cerebro, los modelos de aprendizaje profundo no son modelos del cerebro.
No hay evidencia de que el cerebro implemente algo como los mecanismos de aprendizaje utilizados en modelos modernos de aprendizaje profundo. Para nuestros propósitos, el aprendizaje profundo es un *marco matemático* para las representaciones de aprendizaje de los datos.

Descenso de gradiente

En Machine Learning, los modelos predictivos representan el error que cometen al hacer un predicción mediante una función de **costo.**

Esta función nos dice que tan cerca o parecida es la predicción que el modelo hace en comparación con los valores originales del dataset.

Así a medida que el **costo** de un modelo disminuye, la exactitud (*accuracy*) aumentará, por lo que un valor de costo aproximadamente tendiendo a cero es lo esperado.

Sin embargo, lograr un costo igual a cero puede resultar una tarea dificultosa, a la vez que podríamos estar causando un sobreajuste a los datos de entrenamiento.

El descenso de gradiente es un algoritmo de optimización de los parámetros de un modelo, que permite iteración tras iteración, ajustar dichos parámetros para que la salida del modelo sea lo más exacta o menos *costosa* posible.

Se denomina **descenso de gradiente** ya que utiliza el gradiente de la función costo, para averiguar hacia que lado la función crece, de tal forma de apuntar en sentido opuesto, para saber hacia donde decrece. Sabiendo esto este que se puede derivar la función de costo en función de los parámetros del modelo.

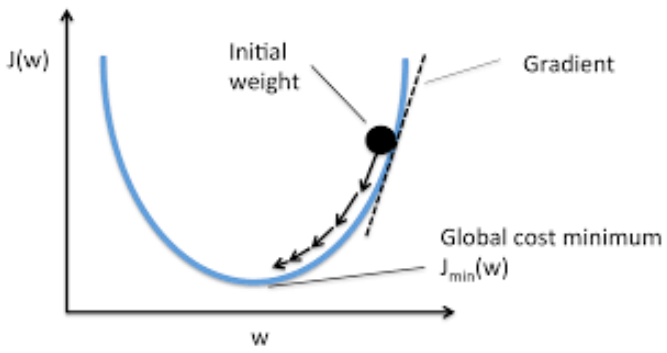
Luego con las derivadas correspondientes a los parámetros podemos actualizar estos mismos, para lograr un nuevo calculo de la función de costo, el cual debería ser menor.

Regresión Lineal

En el caso de la regresión por mínimos cuadrados ordinarios **OLS**, la función de costo es la que se obtiene del promedio de la suma del cuadrado de los residuos. Esta función se denota con la letra mayúscula **J** y equivale al MSE:

$$J = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y} - y)^2$$

El algoritmo inicia valores de parámetros (theta en la figura) de forma *randomizada* y luego en base al cálculo del costo, irá actualizando los valores de dichos parámetros



Descenso de gradiente de la función costo en regresión lineal.

que converjan en el mínimo global
de la función costo.

Entonces, con el cálculo del gradiente en cada iteración, podemos obtener el valor de la derivada con respecto a cada parámetro, de esta forma poder actualizar el valor del parámetro actual, para que nos lleve hacia el menor valor de J. Esto se hace con la formula:

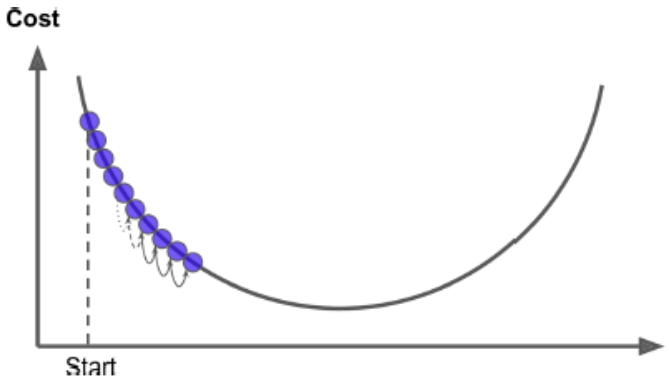
$$w = w0 - \alpha \delta J / \delta w \qquad\qquad b = b0 - \alpha \delta J / \delta b$$

Y así con los nuevos valores de parámetros, se harán nuevas predicciones y se calculará el costo, de esta forma se repite el proceso iterativo hasta que el valor de costo no pueda descender más iteración tras iteración.

Learning Rate

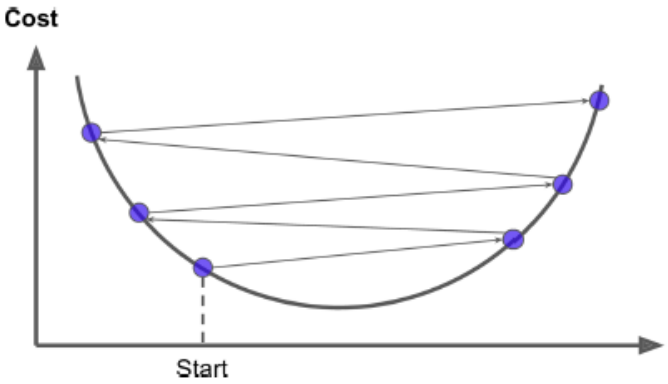
Un hiperparámetro importante en el descenso de gradiente, es la **tasa de aprendizaje** (learning rate) que esta dado por el símbolo α. Este se puede considerar como el *tamaño* de pasos que se considera para alcanzar el punto de convergencia.

Si el learning rate es muy bajo (por ejemplo α=0.0001) el algoritmo tomará muchas iteraciones, ya que cada actualización de parámetros (w,b) será en un valor muy pequeño, lo cual demandará más tiempo para alcanzar el punto de convergencia.



Learning rate demasiado pequeño.

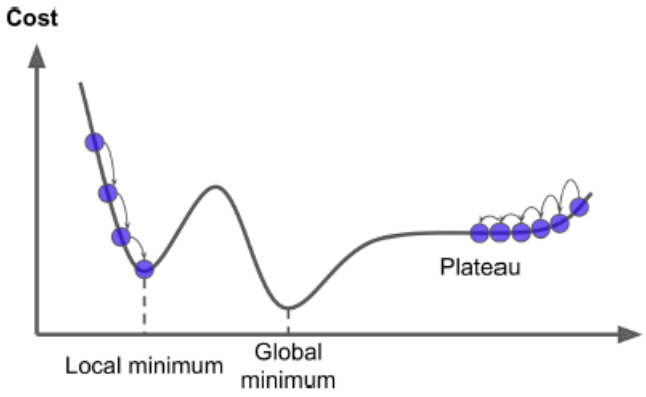
Si elegimos un valor demasiado alto de learning rate, tal vez el algoritmo empiece a diverger, saltando a valores mayores, con lo cual estaríamos alejándonos de encontrar un valor óptimo de la función de costo.



Learning rate demasiado grande.

Afortunadamente, la función de costo de MSE para un modelo de regresión lineal resulta ser una ***función convexa***, lo que significa que si seleccionamos dos puntos en la curva, el segmento de línea que los une nunca cruza la curva. Esto implica que no hay mínimos locales, solo un mínimo global. También es una función continua con una pendiente que nunca cambia abruptamente. Estos dos hechos tienen una gran consecuencia: se garantiza que **Gradient Descent** se acerque arbitrariamente al mínimo global (si esperamos lo suficiente y si la tasa de aprendizaje no es demasiado alta).

Por otro lado, no todas las funciones de costos se ven como cuencos normales y agradables. Puede haber huecos, crestas, mesetas y todo tipo de terrenos irregulares, haciendo muy difícil la tarea de convergencia al mínimo de la función **J**. La siguiente figura muestra los dos desafíos principales con *Gradient Descent*: si la inicialización aleatoria de parámetros inicia el algoritmo a la izquierda, entonces convergerá a un mínimo local, que no es tan bueno como el mínimo global. Si empieza por la derecha, tardará mucho en cruzar la meseta y, si se detiene demasiado pronto, nunca alcanzará el mínimo global.

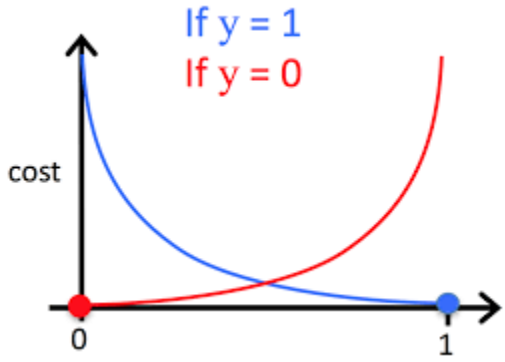


Función de costo real.

Regresión logística

En esta regresión, cómo vimos en la sección de clasificación, la función de costo esta dada por la función de error *binary cross-entropy*, es decir la entropía cruzada binaria entre una clase o la otra [0;1]. Esta función esta representada por los logaritmos que básicamente convierte a la función de costo en una función convexa:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log(h_\theta(x^{(i)})) + (1-y^{(i)})\log(1-h_\theta(x^{(i)}))]$$



Donde *theta θ* representa el espacio de parámetros que modelan la regresión. Por lo tanto para conocer cuanto se ajusta la función costo en función de lo que varia *theta*, usamos derivadas parciales, mediante la **regla de la cadena**, podemos encontrar dicha derivada. La derivada nos dirá cuanto es la pendiente que me lleva hacia el mínimo de la función de costo.

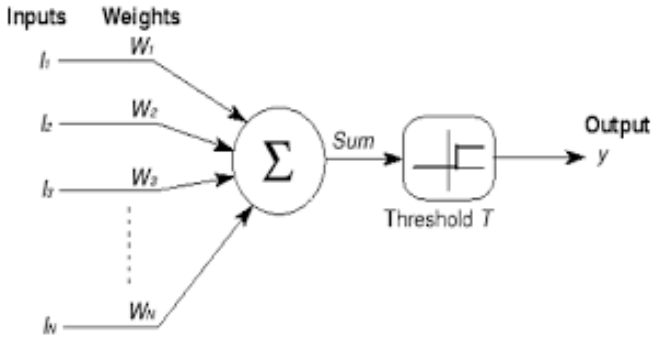
$$\theta = \theta - \alpha * \delta J / \delta \theta$$

Perceptron

Si bien puede parecer que las redes neuronales artificiales son una tecnología de último momento, las **ANN** (por su acrónimo en ingles) existen desde hace bastante tiempo: sus bases fundamentales fueron introducidas por primera vez en 1943 por el neurofisiólogo Warren McCulloch y el matemático Walter Pitts en su artículo "*A Logical Calculus of Ideas Immanent in Nervous Activity*", donde presentaron un modelo computacional simplificado de cómo las neuronas biológicas podrían trabajar juntas en cerebros de animales para realizar cálculos complejos usando lógica proposicional.

> 💡 Debido al carácter "todo o no" de la actividad nerviosa, los eventos neuronales y las relaciones entre ellos pueden tratarse mediante la lógica proposicional. Se encuentra que el comportamiento de cada red se puede describir en estos términos, con la adición de medios lógicos más complicados para las redes que contienen círculos; y que para cualquier expresión lógica que satisfaga ciertas condiciones, uno puede encontrar una red neta de la manera que describe. ["*A Logical Calculus of Ideas Immanent in Nervous Activity*"]

Este modelo representa la unidad fundamental de las redes neuronales, la neurona. La misma representa un cálculo: la suma ponderada de las entradas con sus pesos de cada conexión más un sesgo (*bias*), a dicha suma se le aplica una función no lineal (*umbral*) siguiendo el comportamiento de una neurona biológica y se obtiene una salida. Podemos ver que es muy similar a



Neurona de McCulloch y Pitts.

la matemática detrás de *Regresión Logística*.

El Perceptron está inspirado por la neurona de McCulloch-Pitts y es una de las arquitecturas ANN más simples, inventada en 1957 por Frank Rosenblatt.

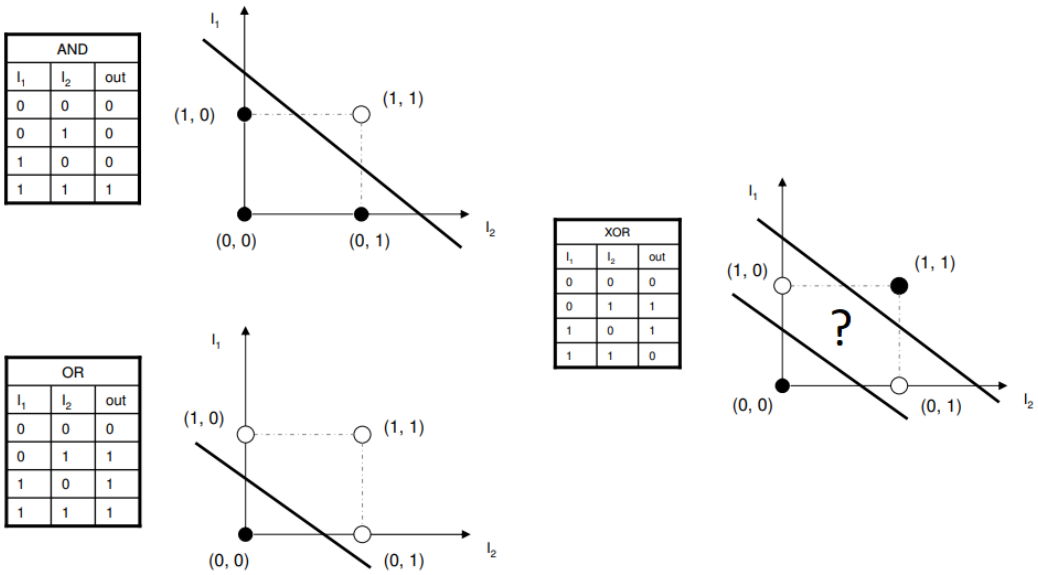
Se basa en una sola **capa** de neuronas artificiales, con cada neurona conectada con todas las entradas. Cada neurona tiene una función de activación de umbral lineal por lo que cada neurona se denomina: **Linear Threshold Unit**. La LTU calcula una suma ponderada de sus entradas, luego aplica una función escalón a esa suma y genera el resultado: .

$$z = w1x1 + w2x2 + \cdots + wnxn = wT\cdot x$$

$$hw(x) = step(z) = step(wT\cdot x) \qquad\qquad step(z) : 1if z \geq 0or0if z < 0$$

> Tener en cuenta que, a diferencia de un clasificador de *regresión logística*, un **perceptrón** no generan una probabilidad de clase; más bien, simplemente hace predicciones basadas en un umbral estricto [0;1]. Esta es una de las buenas razones para preferir la regresión logística al perceptrón.

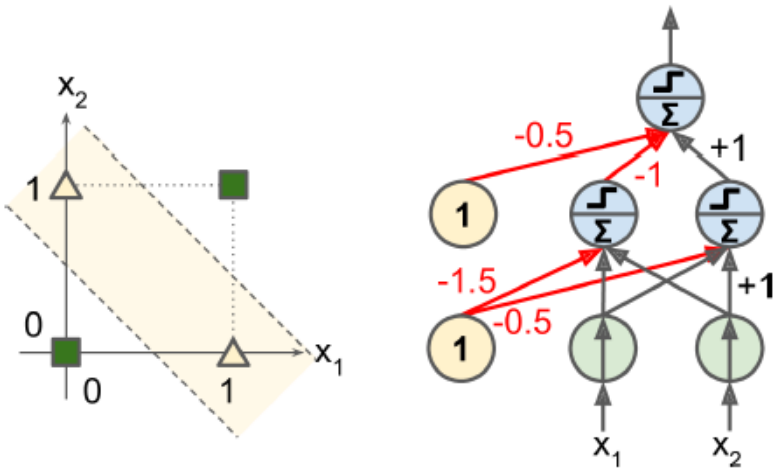
Los perceptrones *simples* poseen varias debilidades graves, en particular el hecho de que son incapaces de resolver algunos problemas triviales (p. ej., el problema de clasificación OR exclusivo (XOR)), esto también es cierto para cualquier otro modelo de clasificación lineal (como en regresión logística), pero en su época, muchos investigadores esperaban mucho más de los perceptrones, como resultado, muchos investigadores abandonaron el estudio de redes neuronales a favor de problemas de nivel superior como la lógica, la resolución de problemas y algoritmos de búsqueda.



Los perceptrones simples son buenos clasificando problemas del tipo AND y OR, pero en problemas de compuerta XOR tienen inconvenientes.

Perceptrón Multicapa

Sin embargo, resulta que algunas de las limitaciones de los perceptrones pueden eliminarse apilando varios perceptrones, y así se puede lograr tener la capacidad para resolver problemas que no son linealmente separables. La ANN resultante se denomina **perceptrón multicapa** (MLP). Como dijimos, un MLP puede resolver el problema XOR, donde para cada combinación de entradas: con entradas (0, 0) o (1, 1) la red genera 0, y con las entradas (0, 1) o (1, 0) genera 1.



Problema XOR resulto por un perceptrón con dos capas.

Una MLP está compuesta de una **capa de entrada**, una o más capas de LTU, llamadas **capas ocultas** y una capa final de LTU llamada **capa de salida**. Cada capa, excepto la capa de salida, incluye una neurona de sesgo y está completamente conectada a la siguiente capa. Cuando una ANN tiene dos o más capas ocultas, se llama **Red Neural Profunda** (Deep Neural Network).

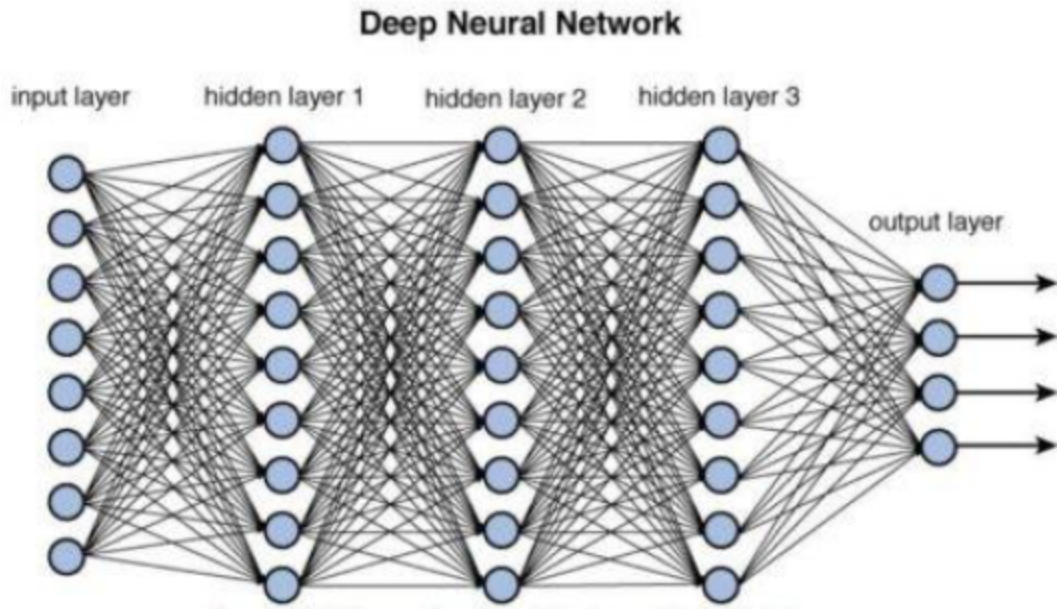


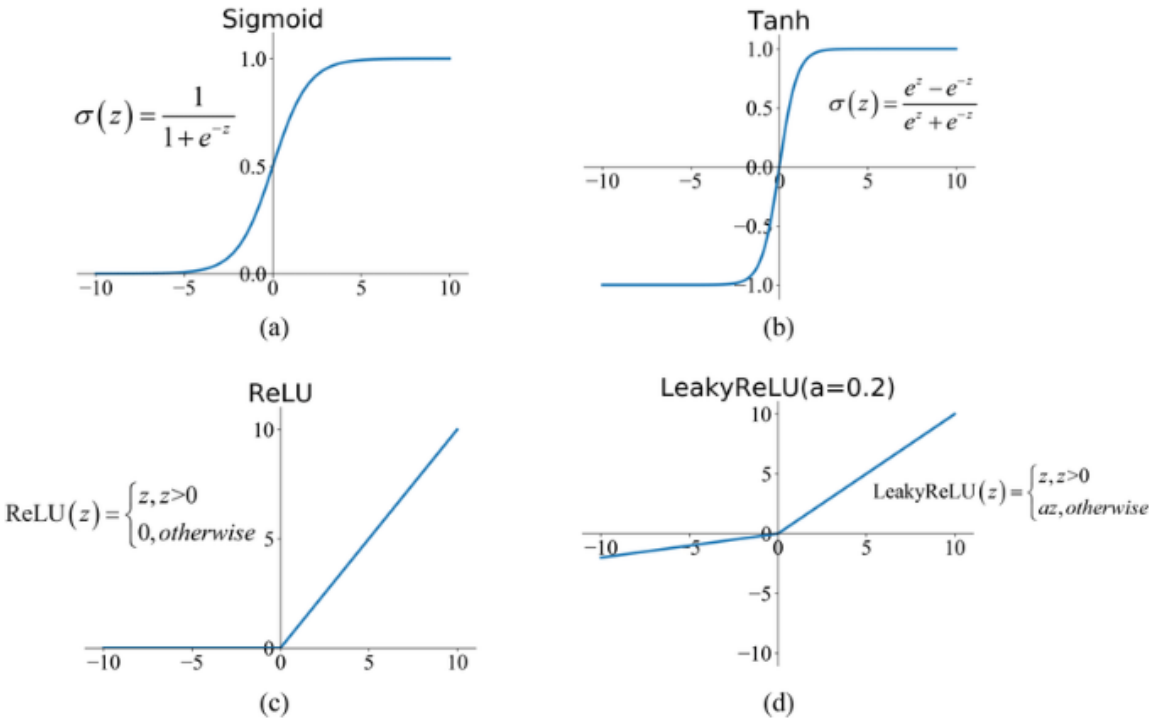
Figure 12.2 Deep network architecture with multiple layers.

Funciones de activación

Algunas funciones de activación más usuales son las que se ven en la figura de abajo. donde por lo general se utiliza **Sigmoid** para función de activación a la salida de una red que tenga como objetivo una tarea de clasificación.

La función **Tanh** se utiliza como activación no lineal de la salida para casos cuando se necesite grandes valores de actualización del descenso de gradiente. Esto es debido a su característica en la derivada.

La función **ReLU** se utiliza como activación de las capas ocultas ya que resulta más rápida para hacer cálculos y el descenso de gradiente no queda atascado en *Plateau*, es decir no entra en la parte de saturación para grandes valores de entrada (lo que si pasa cuando es Sigmoid). Este problema se conoce en aprendizaje profundo como *Vanishing Gradient.*



ReLU tiene el problema de que cualquier entrada *negativa* dada a la función de activación, la convertirá en valor cero, lo que disminuye la capacidad del modelo para ajustarse o entrenarse a partir de los datos correctamente.
La **LeakyReLU** nos ayuda a aumentar el alcance de la función ReLU. Por lo general, el valor de ***a*** es aproximadamente 0,01 y cuando no es 0,01, se denomina **Randomized ReLU**. Por lo tanto, el rango de Leaky ReLU es (-∞ ; ∞).

Optimizadores

Como vimos anteriormente en descenso de gradiente, cada neurona realiza una función matemática que transforma los datos de entrada:

$$z(X) = w * X + b$$

En una red neuronal, *w* y *b* son tensores que resultan ser *atributos* de la capa (si consideramos ambos como vectores). Estos parámetros o *pesos*, son lo que la red aprende durante el entrenamiento.

AL inicio del entrenamiento estos *pesos* se inician de forma aleatoria (por lo general w de forma estandarizada y b con ceros). En este punto las predicciones no son significativas ya que el modelo aún no aprende a representar los datos.

Posteriormente, sigue un proceso de ajuste de estos parámetros mediante un proceso que se conoce como **Backpropagation**, esta es la forma de *aprendizaje* que utiliza la red neuronal.

Esto ocurre en un proceso iterativo en forma de *loop:*

1. Seleccionar un conjunto de muestras de X e y.
2. Introducir los datos de X en la red y hacer una predicción. Esto se conoce como **Forward Pass.**
3. Calcular el error de la red neuronal en la predicción con dicho conjunto de muestras.
4. Actualizar los *pesos* (parámetros) de la red en un **Backpropagation** para reducir el error en la siguiente iteración con un nuevo conjunto seleccionado.

Eventualmente se logrará tener una red neuronal que nos muestra un bajo valor de *loss* en el entrenamiento, lo cual podemos decir que la misma *ha aprendido*.

A simple vista parece que todo esto proceso es bastante simple, **pero**, como se puede saber cuánto hay que modificar un *peso* de la red para que aprende a predecir correctamente?

Una forma sería ajustar los valores de forma manual en cada iteración, pero esto resultaría *extremadamente ineficiente*.

Un enfoque más óptimo es tomar ventaja de que todas las operaciones usadas por la red neuronal son *diferenciables* y calculamos el *gradiente* del *loss* con respecto a los *pesos* de la red. Se puede mover los *pesos* en la dirección opuesta al gradiente, a fin de minimizar el *loss*.

Stocastic Gradient Descent

Dada una función diferenciable, es teóricamente posible encontrar el mínimo analíticamente, se sabe que el mínimo de una función es un punto donde la derivada se hace cero, a ese punto es donde queremos llegar, ya que será el mínimo valor que tome la función de *loss.*

Esto como dijimos anteriormente se aplica con el cálculo del gradiente de la función de *costo*, que por medio de la regla de la cadena podemos calcular las derivadas parciales con respecto a cada parámetro.

En la realidad cuando la red neuronal tiene una neurona es decir existen como mínimo 2 parámetros a obtener resulta viable el cálculo del *Descenso de Gradiente*, pero en una red neuronal profunda, el mismo resulta inviable para todo el conjunto de datos.

Entonces lo que se hace es trabajar con grupos de datos por iteración (*batches en ingles*) armados de forma aleatoria. Esto es lo que se conoce como **Mini-Batch Stocastic Gradiente Descent** (Mini-batch SGD). El término estocástico es por la naturaleza aleatoria como se toman los conjuntos de datos.

Una variante del **Mini-batch SGD** es tomar un conjunto de características y etiqueta (X,y) por cada iteración, en vez de tomar conjuntos de datos. Esto se conoce como *true* **SGD**.

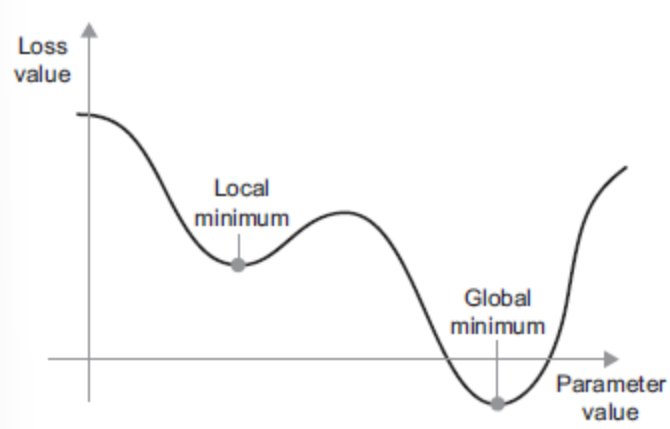
En contrapartida, se puede tomar *todos* los datos en cada iteración con el método **Batch SGD**. Este caso de los 3 resulta el que mejor exactitud presentará, pero será mas costoso en cuanto a poder de computación.

Ningún extremo es bueno, por un lado el **SGD** resulta muy fácil de calcular para la red pero con métricas no muy buenas, el **Batch SGD** presenta métricas muy buenas pero requiere mayor poder de computo. Por lo general, se utiliza la técnica de **Mini-batch SGD** con batches de tamaño de potencias de dos: 4, 8, 16, 32, 64, 128.

Existen otras variantes del **SGD** que tienen en cuenta los *pesos* anteriores que se actualizaron para calcular. Todos estos se conocen como métodos de optimización o simplemente ***Optimizadores***. Algunos de estos pueden ser:

- Momentum.
- RMSprop.
- Adagrad.
- Adam.

Estos se encargan tanto de la convergencia del mínimo global como de la velocidad del descenso.



En la gráfica podemos ver que alrededor de cierto valor de parámetro, hay un *mínimo local.* Alrededor de ese punto, moverse hacia la izquierda daría como resultado un aumento de la pérdida, pero también lo haría hacia la derecha. Si este parámetro se optimizara usando **SGD**, entonces el proceso de optimización se atascaría en el mínimo local en vez de alcanzar al *mínimo global*.

Se puede evitar este problema utilizando ***momentum***, que se inspira en la física. Una imagen mental útil es pensar en el proceso de optimización como una pequeña bola que rueda por la curva de pérdida. Si tiene suficiente impulso, la bola no se atascará en un barranco y terminará en el *mínimo global*. El *momentum* se implementa moviendo la pelota en cada paso basándose no solo en el valor de pendiente (gradiente) actual (aceleración actual) sino también en la velocidad actual (resultante de la aceleración pasada). En la práctica, **momentum** significa que al actualizar el parámetro w nos basamos no solo en el valor del gradiente actual sino también en la actualización del parámetro anterior.

Referencias

1. Géron, A. (2019). *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. O'Reilly Media.
2. VanderPlas, J. (2016). *Python data science handbook: Essential tools for working with data*. " O'Reilly Media, Inc."
3. Chollet, F. (2021). *Deep learning with Python*. Simon and Schuster.

4. Certificación Universitaria en Data Science. Mundos E - Universidad Nacional de Cordoba.

Made with ❤️ Ignacio Bosch