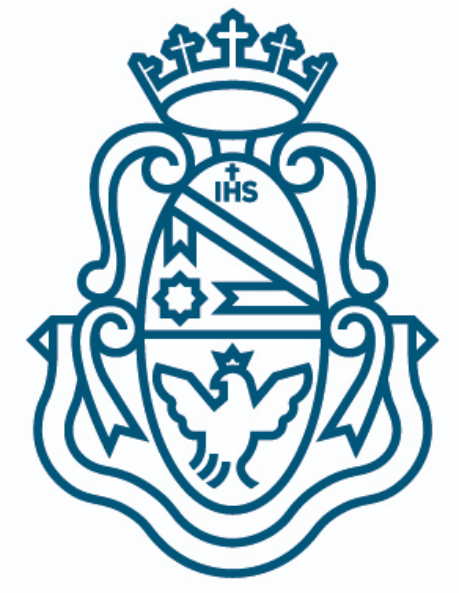
Universidad Nacional de Córdoba
Facultad de Ciencias Exactas, Físicas y Naturales



Prácticas Profesionales Supervisadas
Primeras 100 hs.

Capacitación de Machine Learning con Redes Neuronales Convolucionales aplicado a imágenes

Gerardo Collante

supervisado por
Ing. Aldo Algorry

19 de febrero de 2020

Índice

1. Introducción

1.1. Objetivos de las PPS

En mis prácticas profesionales supervisadas puse mi foco en el *aprendizaje de máquina* o como se conoce comúnmente, *Machine Learning*, debido a la extensa diversidad de posibilidades que brinda esta tecnología, teniendo por seguridad que en un futuro ser un *commodity* indispensable en cualquier empresa que esta relacionada con el campo de IT aunque no sea su campo de acción principal.

Por ello decidí realizar mis prácticas contribuyendo a un proyecto de investigación doctoral con el fin de aprender la tecnología.

1.2. Definiciones

En las primeras 100 horas el objetivo fue instruirme sobre la Inteligencia Artificial (IA) ya que no posea ningún conocimiento en el campo. El objetivo final eran las Redes Neuronales Convolucionales, *Convolutional Neural Networks, (CNN)* pero ante debía instruirme sobre Redes Neuronales, *Neural Networks (NN)*.

> La *Inteligencia Artificial (Artificial Intelligence)* se define como el estudio de los .ªgentes inteligentes", i.e. cualquier dispositivo que perciba su entorno y tome medidas que maximicen sus posibilidades de lograr con éxito sus objetivos.
>
> Poole et al. [1]

Esta definición nos da la idea de que la IA es un sistema reactivo, que reacciona a cambios externos y actúa en consecuencia, sin embargo, todavía no queda claro el rol del *Aprendizaje de Máquina*.

> El *aprendizaje automático (Machine Learning)* es el estudio científico de algoritmos y modelos estadísticos que los sistemas informáticos utilizan para realizar una tarea específica sin utilizar instrucciones explícitas, sino que se basan en patrones e inferencia. Es visto como un subcampo de inteligencia artificial. Los algoritmos de aprendizaje automático crean un modelo matemático basado en datos de muestra, conocidos como "datos de entrenamiento", para hacer predicciones o decisiones sin ser programado explícitamente para realizar la tarea.
>
> Bishop [2]

Aquí se define al ML como una parte de la AI. El siguiente diagrama ayudará a aclarar conceptos [3].

1. Los datos etiquetados o *labeled data* será el set de entrenamiento o *dataset* que entrenará el modelo predictivo.
2. Éste se compone de capas de redes neuronales, que serán entrenadas para un fin determinado, *e.g.* para clasificar los datos en correctos o incorrectos.
3. Una vez entrenado el modelo pueden ingresar nuevos datos o *new data* para ser clasificados.

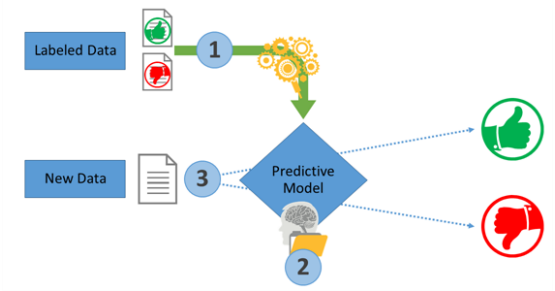


Figura 1: Diagrama simple de modelo de *Machine Learning*.

2. Aprendizaje

La bibliografia elegida para mi aprendizaje sobre redes neuronales fue *"Make your own neural network"* [4] del Dr. Tariq Rashid, debido a que muestra como crear desde un punto de vista matemático una red neuronal desde cero (*scratch*), para luego ser codificada en *Python*.

2.1. Repaso y conceptos básicos

En los primeros capítulos del libro se dan conceptos de funciones y como los métodos iterativos pueden ser utilizados para resolver problemas simples. Queda a cargo del lector revisar la bibliografía para una mayor comprensión del tema en caso que estos conceptos no estén claros. Para un punto de vista más matemático y conciso será utilizada la bibliografía *"Deep Learning for Computer Vision with Python"* [5] del Dr. Rosebrock.

3. Redes neuronales

La palabra neuronal es la forma adjetiva de "neurona", y red denota una estructura tipo grafo; por lo tanto, una *Red Neuronal Artificial* es un sistema de computación que intenta imitar (o al menos, está inspirado en) las conexiones neuronales en nuestro sistema nervioso. Las redes neuronales artificiales también se denominan *redes neuronales* o *sistemas neuronales artificiales*. Es común abreviar la red neuronal artificial y referirse a ellas como "NN".

Para que un sistema se considere un NN, debe contener una estructura de grafo dirigida y etiquetada donde cada nodo del gráfico realice un cálculo simple. Según la teoría de grafos, sabemos que un gráfico dirigido consiste en un conjunto de nodos (es decir, vértices) y un conjunto de conexiones (es decir, bordes) que unen pares de nodos.

- Las entradas ingresan a la red.
- Cada conexión lleva una señal a través de las dos capas ocultas en la red.
- Una función final calcula la etiqueta de clase de salida.

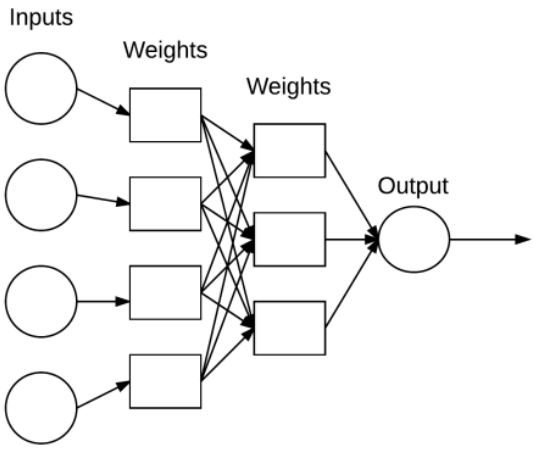


Figura 2: Arquitectura simple de red neuronal.

Cada nodo realiza un cálculo simple. Cada conexión transporta una señal (es decir, la salida del cálculo) de un nodo a otro, marcada por un peso que indica el grado en que la señal se amplifica o disminuye. Algunas conexiones tienen grandes pesos positivos que amplifican la señal, lo que indica que la señal es muy importante al hacer una clasificación. Otros tienen pesos negativos, lo que disminuye la intensidad de la señal, lo que especifica que la salida del nodo es menos importante en la clasificación final. Llamamos a dicho sistema una Red Neuronal Artificial si consta de una estructura de grafo (como en la Figura 2) con pesos de conexión que se pueden modificar utilizando un algoritmo de aprendizaje.

3.1. Relación con la biología

Nuestros cerebros están compuestos por aproximadamente 10 mil millones de neuronas, cada una conectada a unas 10,000 otras neuronas. El cuerpo celular de la neurona se llama soma, donde las entradas (dendritas) y las salidas (axones) conectan el soma con otro (Figura 3).

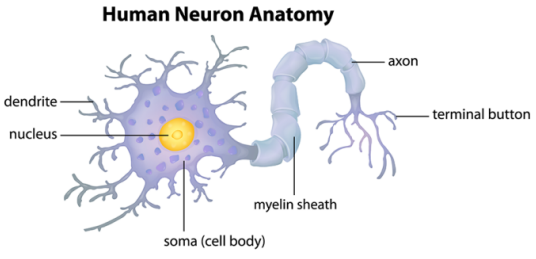


Figura 3: Estructura de una neurona biológica.

Cada neurona recibe entradas electroquímicas de otras neuronas en sus den-

dritas. Si estas entradas eléctricas son lo suficientemente potentes como para activar la neurona, entonces la neurona activada transmite la señal a lo largo de su axón, transmitiéndola a las dendritas de otras neuronas. Estas neuronas unidas también pueden activarse, continuando así el proceso de transmitir el mensaje. La conclusión clave aquí es que el disparo de una neurona es una operación binaria: la neurona se dispara o no se dispara. No hay diferentes "grados"de disparo. En pocas palabras, una neurona solo se disparará si la señal total recibida en el soma excede un umbral dado. Sin embargo, tenga en cuenta que los ANN simplemente se inspiran en lo que sabemos sobre el cerebro y cómo funciona. El objetivo del aprendizaje profundo no es imitar cómo funcionan nuestros cerebros, sino tomar las piezas que entendemos y permitirnos trazar paralelos similares en nuestro propio trabajo.

3.2. Modelos artificiales

Comencemos por ver un NN básico que realiza una suma ponderada simple de las entradas o *inputs* en la Figura 4. Los valores x_1, x_2 y x_3 son las *inputs* a nuestro NN y generalmente corresponden a una sola fila (es decir, punto de datos) de nuestra matriz de diseño. El valor constante 1 es nuestro sesgo o *bias* que se supone incrustado en la matriz de diseño. Podemos pensar en estas *inputs* como los vectores de características o *features* de entrada a la NN.

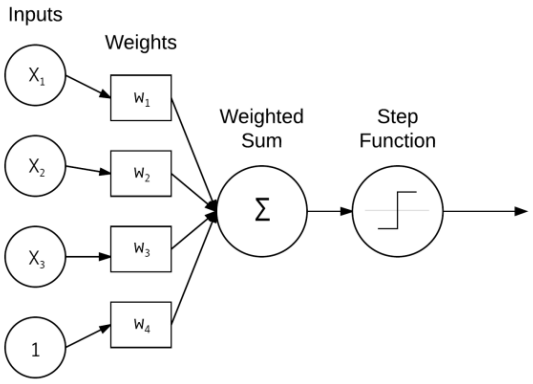


Figura 4: Simple NN.

Cada x está conectada a una neurona a través de un vector de peso W que consiste en $w_1, w_2, \ldots w_n$, lo que significa que para cada entrada x también tenemos un peso asociado w. Finalmente, el nodo de salida a la derecha toma la suma ponderada, aplica una función de activación f (utilizada para determinar si la neurona se "dispara.º no) y genera un valor. Expresando la salida matemáticamente, normalmente encontrarás las siguientes tres formas:

- $f(w_1x_1 + w_2x_2 + \cdots + w_nx_n)$
- $f(\sum_{i=1}^{n} w_ix_i)$
- O $f(net)$, donde $net = \sum_{i=1}^{n} w_ix_i$

3.3. Funciones de activación

La función de activación más simple es la "función de paso", utilizada por el algoritmo Perceptron.

$$f(net) = \begin{cases} 1 & \text{si } net > 0 \\ 0 & \text{si } net \leq 0 \end{cases}$$

Esta es una función de umbral muy simple, sin embargo, aunque es fácil de usar e intuitiva, no es diferenciable, lo cual puede llevar a problemas cuando apliquemos el descenso por gradiente. Por ello se presentan en la Figura 5 diferentes tipos de funciones de activación con sus respectivos gráficos.

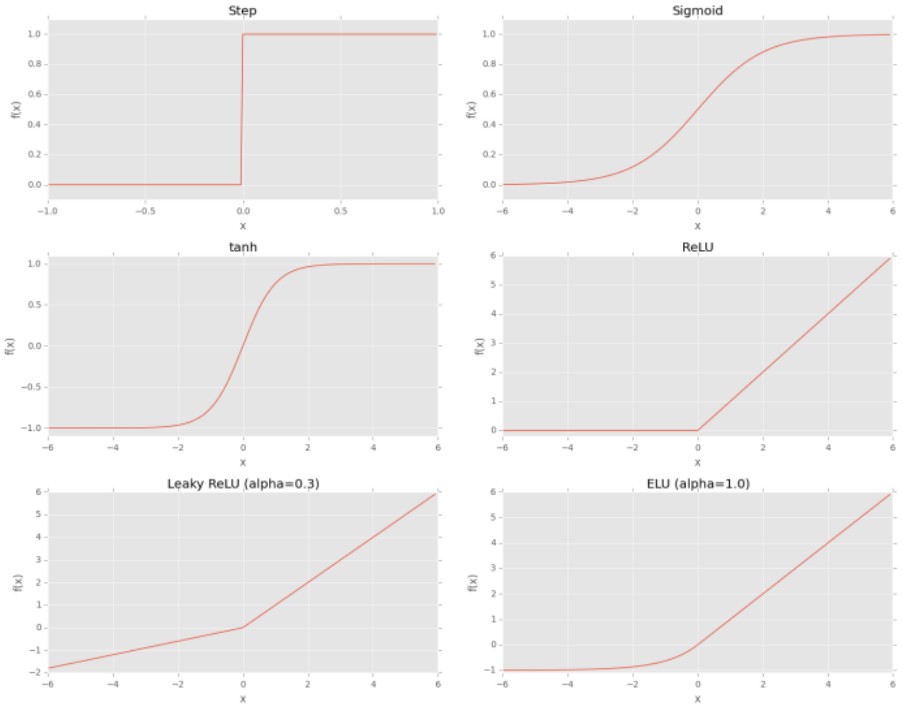


Figura 5: **Arriba-izquierda**: Función escalón. **Arriba-derecha**: Función sigmoidea. **Medio-izquierda**: Tangente hiperbólica. **Medio-derecha**: activación ReLU (función activación más usada en *Deep Learning*). **Abajo-izquierda**: Leaky ReLU, variante de ReLU que permite valores negativos. **Abajo-derecha**: ELU, otra variante de ReLU que obtiene mejor performance que Leaky ReLU.

Una de las funciones de activación más usadas en la historia de la literatura de NN es la función sigmoidea, que sigue la siguiente ecuación:

$$t = \sum_{i=1}^{n} w_i x_i \quad s(t) = \frac{1}{1 + e^{-t}} \tag{1}$$

La función sigmoidea es una mejor opción para el aprendizaje que la función de paso simple, ya que:

1. Es continua y diferenciable en todas partes.

2. Es simétrica alrededor del eje y.
3. Se acerca asintóticamente a sus valores de saturación.

La principal ventaja aquí es que la suavidad de la función sigmoidea hace que sea más fácil diseñar algoritmos de aprendizaje. Sin embargo, hay dos grandes problemas con la función sigmoidea:

1. Las salidas del sigmoide no están centradas en cero.
2. Las neuronas saturadas esencialmente eliminan el gradiente, ya que el delta del gradiente será extremadamente pequeño.

La tangente hiperbólica, o tanh (con una forma similar del sigmoide) también se usó fuertemente como una función de activación hasta fines de la década de 1990. La ecuación para tanh sigue:

$$f(z) = tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{2}$$

La función tanh está centrada en cero, pero los gradientes aún se eliminan cuando las neuronas se saturan. Ahora sabemos que hay mejores opciones para las funciones de activación que las funciones sigmoide y tanh. Específicamente, la Unidad Lineal Rectificada (ReLU), definida como:

$$f(x) = max(0, x) \tag{3}$$

Las ReLU también se denominan "funciones de rampa"debido a cómo se ven cuando se trazan. La función es cero para entradas negativas pero luego aumenta linealmente para positivas valores. La función ReLU no es saturable y también es extremadamente eficiente computacionalmente. Empíricamente, la función de activación ReLU tiende a superar a las funciones sigmoide y tanh en casi todas las aplicaciones. Sin embargo, surge un problema cuando tenemos un valor de cero: no se puede tomar el gradiente.

3.4. Arquitecturas de redes *feedforward*

Si bien hay muchas, muchas arquitecturas NN diferentes, la arquitectura más común es la red hacía adelante o *feedforward*, como se presenta en la Figura 6.

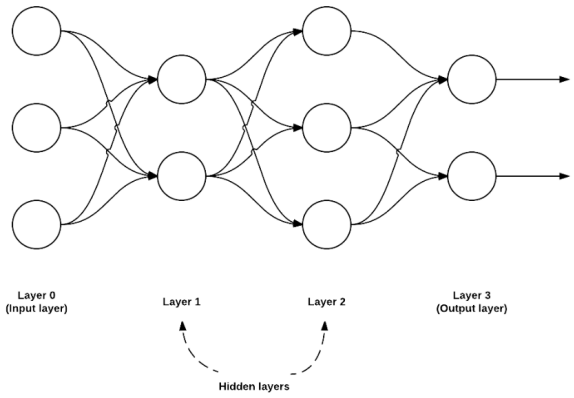


Figura 6: Un ejemplo de una red neuronal *feedforward*.

En este tipo de arquitectura, solo se permite una conexión entre los nodos de los nodos en la capa i a los nodos en la capa $i+1$ (de ahí el término *feedforward*). No hay conexiones hacia atrás o entre capas permitidas. Cuando las redes de retroalimentación incluyen conexiones de retroalimentación (conexiones de salida que retroalimentan las entradas) se denominan redes neuronales recurrentes.

Para describir una red *feedforward*, normalmente usamos una secuencia de enteros para depositar rápida y concisamente el número de nodos en cada capa. Por ejemplo, la red en la Figura 10.5 anterior es una red de alimentación directa 3-2-3-2:

- La capa 0 contiene 3 entradas, nuestros valores x_i. Estos podrían ser intensidades de píxeles sin procesar de una imagen o un vector de características extraído de la imagen.
- Las capas 1 y 2 son capas ocultas que contienen 2 y 3 nodos, respectivamente.
- La capa 3 es la capa de salida o la capa visible: allí es donde obtenemos la clasificación de salida general de nuestra red. La capa de salida generalmente tiene tantos nodos como etiquetas de clase; un nodo para cada salida potencial.

3.5. Redes multicapa

Las redes multicapas, *i.e.* con varias capas de neuronas pueden ser modeladas matemáticamente como se muestra a continuación. Supongamos W como la matriz de pesos y el vector b como el vector sesgo o *bias*. Consideremos:

$$z(x) = Wx + b = \sum_{i=1}^{n} w_i x_i + b \tag{4}$$

Además cabe mencionar que la multiplicación punto a punto entre dos matrices de igual dimensión es lo que se conoce como el producto Hadamard.

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{5}$$

Por último definimos la salida de nuestro modelo como:

$$\hat{y} = \sigma(\sum_{i=1}^{n} w_i x_i + b) \tag{6}$$

3.6. Función pérdida

El objetivo del algoritmo de descenso de gradiente es minimizar la función de costo para que nuestro modelo neuronal pueda aprender. Pero antes debemos definir que es la función costo o pérdida [6]. En el cálculo, los máximos (o mínimos) de cualquier función pueden ser descubiertos por:

1. Tomando la derivada de primer orden de la función e igualándola a 0. El punto encontrado de esta manera puede ser el punto de máximo o mínimo.

2. Sustituimos estos valores (el punto que acabamos de encontrar) en la derivada de segundo orden de la función y si el valor es positivo, *i.e.*> 0, entonces ese punto (s) representa el punto (s) de mínimos locales o máximos locales.

Necesitamos cerrar la brecha entre la salida del modelo y la salida real. Cuanto menor sea la brecha, mejor será nuestro modelo en sus predicciones y más confianza mostrará al predecir.

La **función de pérdida o costo** esencialmente modela la diferencia entre la predicción de nuestro modelo y la salida real. Idealmente, si estos dos valores están muy separados, el valor de pérdida o el valor de error deberían ser mayores. Del mismo modo, si estos dos valores están más cerca, el valor del error debería ser bajo. Una posible función de pérdida podría ser:

$$J(\Theta) = \hat{y} - y / y \in \{0, 1\} \tag{7}$$

Pero, en lugar de tomar esta función como nuestra función de pérdida, terminamos considerando la siguiente función:

$$J(\Theta) = \frac{\|\hat{y} - y\|^2}{2} \tag{8}$$

Esta función se conoce como error al cuadrado . Simplemente tomamos la diferencia entre la salida real y y la salida predicha $\hat{y}$ elevamos al cuadrado ese valor (de ahí el nombre) y lo dividimos entre 2.

Una de las principales razones para preferir el error al cuadrado en lugar del error absoluto es que el error al cuadrado es diferenciable en todas partes , mientras que el error absoluto no lo es (su derivada no está definida en 0).

Además, los beneficios de la cuadratura incluyen:

- La cuadratura siempre da un valor positivo , por lo que la suma no será cero.
- Hablamos de suma aquí porque sumaremos los valores de pérdida o error para cada imagen en nuestro conjunto de datos de entrenamiento y luego haremos un promedio para encontrar la pérdida para todo el lote de ejemplos de entrenamiento.
- La cuadratura enfatiza las diferencias más grandes, una característica que resulta ser buena y mala.

La función de error que usaremos aquí se conoce como el error cuadrático medio y la fórmula es la siguiente:

$$J(\Theta) = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y}_i - y_i)^2 \tag{9}$$

Calculamos el error al cuadrado para cada *feature* en nuestro *dataset* y luego encontramos el promedio de estos valores y esto representa el error general del modelo en nuestro conjunto de entrenamiento.

Consideremos el ejemplo de una sola *feature* con solo 2 características de antes. Dos características significan que tenemos 2 valores de peso correspondientes y un valor de sesgo. En total, tenemos 3 parámetros para nuestro modelo.

$$\hat{y} = w_1 x_1 + w_2 x_2 + b \tag{10}$$

$$J(\Theta) = \frac{1}{2m} \sum_{i=1}^{m} (w_1 x_1^{(i)} + w_2 x_2^{(i)} + b - y^{(i)})^2 \tag{11}$$

Queremos encontrar valores para nuestros pesos y el sesgo que minimiza el valor de nuestra función de pérdida. Dado que esta es una ecuación de múltiples variables, eso significa que tendríamos que tratar con derivadas parciales de la función de pérdida correspondiente a cada una de nuestras variables w_1, w_2 y b.

$$\frac{\partial J}{\partial w_1} \frac{\partial J}{\partial w_2} \frac{\partial J}{\partial b} \tag{12}$$

Esto puede parecer lo suficientemente simple porque solo tenemos 3 variables diferentes. Sin embargo tenemos tantos pesos como features *i.e.* w_n pesos.

Hacer una optimización multivariante con tantas variables es computacionalmente ineficiente y no es manejable. Por lo tanto, recurrimos a alternativas y aproximaciones.

3.7. Descenso de gradiente

Este es el algoritmo que ayuda a nuestro modelo a aprender . Sin la capacidad de aprendizaje, cualquier modelo de aprendizaje automático es esencialmente tan bueno como un modelo de adivinanzas al azar.

Es la capacidad de aprendizaje que otorga el algoritmo de descenso de gradiente lo que hace que el aprendizaje automático y los modelos de aprendizaje profundo funcionen.

El objetivo de este algoritmo es minimizar el valor de nuestra función de pérdida y queremos hacer esto de manera eficiente.

Como se discutió anteriormente, la forma más rápida sería encontrar derivadas de segundo orden de la función de pérdida con respecto a los parámetros del modelo. Pero, eso es computacionalmente costoso.

La intuición básica detrás del descenso del gradiente puede ilustrarse mediante un escenario hipotético.

Una persona está atrapada en las montañas y está tratando de bajar (es decir, tratando de encontrar los mínimos). Hay mucha niebla de tal manera que la visibilidad es extremadamente baja. Por lo tanto, el camino hacia abajo de la montaña no es visible, por lo que deben usar la información local para encontrar los mínimos.

Pueden usar el método de descenso en gradiente, que consiste en mirar la inclinación de la colina en su posición actual, luego proceder en la dirección con el descenso más empinado (es decir, cuesta abajo).

Si trataban de encontrar la cima de la montaña (es decir, los máximos), entonces avanzarían en la dirección con el ascenso más empinado (es decir, cuesta arriba). Usando este método, eventualmente encontrarían su camino.

Sin embargo, suponga también que la pendiente de la colina no es inmediatamente obvia con una simple observación, sino que requiere un instrumento sofisticado para medir, que la persona tiene en ese momento.

Se necesita bastante tiempo para medir la inclinación de la colina con el instrumento, por lo tanto, deben minimizar el uso del instrumento si quieren bajar la montaña antes del atardecer.

La dificultad es elegir la frecuencia con la que deben medir la inclinación de la colina para no desviarse.

En esta analogía:

- La persona representa nuestro **algoritmo de aprendizaje**, y el camino que baja por la montaña representa la **secuencia de actualizaciones de parámetros** que nuestro modelo eventualmente explorará.
- La inclinación de la colina representa la **pendiente de la superficie de error en ese punto**.
- El instrumento utilizado para medir la inclinación es la **diferenciación** (la pendiente de la superficie de error se puede calcular tomando la derivada de la función de error al cuadrado en ese punto). Esta es la aproximación que hacemos cuando aplicamos el descenso de gradiente. Realmente no sabemos el punto mínimo, pero sí sabemos la dirección que nos llevará a los mínimos (locales o globales) y damos un paso en esa dirección.
- La dirección en la que la persona elige viajar se alinea con el gradiente de la superficie de error en ese punto.
- La cantidad de tiempo que viajan antes de tomar otra medida es la **velocidad de aprendizaje del algoritmo**. Esto es esencialmente lo importante que nuestro modelo (o la persona que va cuesta abajo) decide dar cada vez.

La Figura 7 es una de las más comunes asociadas con el descenso de gradiente y muestra que nuestra función de error es una función convexa suave. También muestra que el algoritmo de descenso de gradiente encuentra el mínimo global.

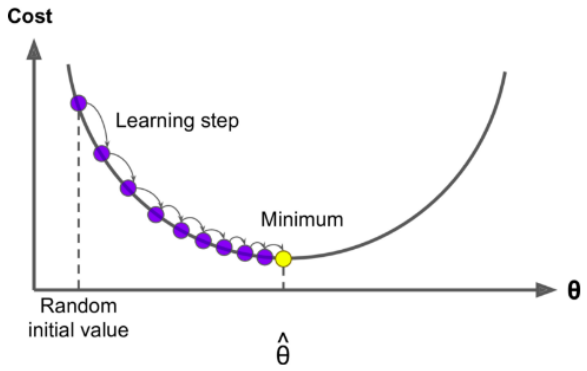


Figura 7: Representación gráfica del descenso de gradiente.

Por tanto para actualizar la matriz de pesos y de sesgo serán utilizadas las siguientes ecuaciones.

$$W' = W - \alpha \frac{\partial J}{\partial W} \tag{13}$$

$$b' = b - \alpha \frac{\partial J}{\partial b} \tag{14}$$

El α representa la tasa de aprendizaje o *learning rate* para nuestro algoritmo de pendiente de descenso, *i.e.* el tamaño de paso para ir cuesta abajo.

3.8. Backpropagation

Ya sabemos cómo fluyen las activaciones en la dirección hacia adelante. Tomamos las *features* de entrada, las transformamos linealmente, aplicamos la activación sigmoidea en el valor resultante y finalmente tenemos nuestra activación que luego usamos para hacer una predicción.

Lo que veremos en esta sección es el flujo de gradientes a lo largo de la línea roja en la Figura 8 mediante un proceso conocido como retropropagación o *backpropagation*, que es esencialmente la regla de la cadena de cálculo aplicada a los gráficos computacionales.

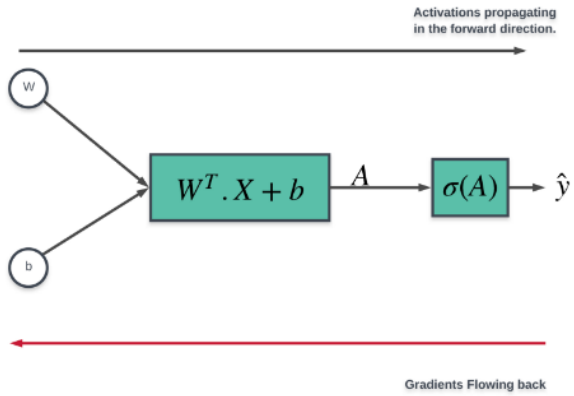


Figura 8: Las activaciones se propagan hacía adelante, pero los gradientes fluyen hacía atrás.

Digamos que queríamos encontrar la derivada parcial de la variable y con respecto a x de la Figura 9. No podemos descubrirlo directamente porque hay otras 3 variables involucradas en el gráfico computacional. Entonces, hacemos este proceso iterativamente yendo hacia atrás en el gráfico de cálculo.

Primero descubrimos la derivada parcial de la salida y con respecto a la variable C. Luego usamos la regla de la cadena de cálculo y determinamos la derivada parcial con respecto a la variable B y así sucesivamente hasta que obtengamos la derivada parcial que estamos buscando.

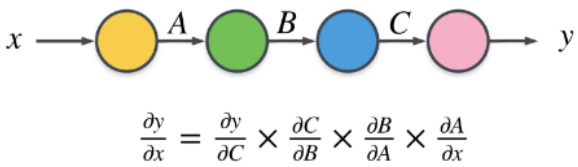


Figura 9: Representación de grafo simple.

Utilizando la función pérdida definida en la ecuación 9 y reescribiéndola en su forma vectorial.

$$J(\Theta) = \frac{1}{2}\|\hat{Y} - Y\|^2 \tag{15}$$

La derivada parcial de la función de pérdida con respecto a la activación de nuestro modelo es:

$$\frac{\partial J}{\partial \hat{Y}} = \frac{1}{2}\frac{\partial J}{\partial \hat{Y}}\|\hat{Y} - Y\|^2 = \frac{1}{2}2\hat{Y} - Y\frac{\partial}{\partial \hat{Y}}(\hat{Y} - Y) = (\hat{Y} - Y)\frac{\partial}{\partial \hat{Y}}\|\hat{Y} - Y\| = (\hat{Y} - Y) \tag{16}$$

Avancemos un paso hacia atrás y calculemos nuestra próxima derivada parcial. Esto nos llevará un paso más cerca de los gradientes reales que queremos calcular.

Este es el punto donde aplicamos la regla de la cadena que mencionamos antes. Entonces, para calcular la derivada parcial de la función de pérdida con respecto a la salida transformada lineal, es decir, la salida de nuestro modelo antes de aplicar la activación sigmoidea:

$$\frac{\partial J}{\partial D} = \frac{\partial J}{\partial \hat{Y}}\frac{\partial \hat{Y}}{\partial A} \tag{17}$$

La primera parte de esta ecuación es el valor que habíamos calculado en la ecuación 16. Lo esencial para calcular aquí es la derivada parcial de la predicción de nuestro modelo con respecto a la salida transformada linealmente.

Veamos la ecuación para la predicción de nuestro modelo, la función de activación sigmoidea.

$$\hat{Y} = \sigma(A) = \frac{1}{1 + e^{-A}} \tag{18}$$

Derivada de la salida final de nuestro modelo, *i.e.* significa la derivada parcial de la función sigmoide con respecto a su entrada.

$$\frac{\partial}{\partial A}\sigma(A) = \frac{\partial}{\partial A}\frac{1}{1 + e^{-A}} = \frac{\partial}{\partial A}(1 + e^{-A})^{-1} \tag{19}$$

$$-1(1 + e^{-A})^{-2}\frac{\partial}{\partial A}(1 + e^{-A}) = -1(1 + e^{-A})^{-2}(-e^{-A}) = \frac{e^{-A}}{(1 + e^{-A})^2} \tag{20}$$

Continuando, podemos simplificar aún más esta ecuación.

$$\sigma(A) = \frac{1}{1 + e^{-A}} \tag{21}$$

$$e^{-A} = \frac{1}{\sigma(A)} - 1 = \frac{1 - \sigma(A)}{\sigma(A)} \tag{22}$$

Substituyendo este valor en la ecuación 17 obtenemos:

$$\frac{\partial J}{\partial A} = \frac{\partial J}{\partial \hat{Y}} \frac{e^{-A}}{(1 + e^{-A})^2} \tag{23}$$

$$\frac{\partial J}{\partial A} = \frac{\partial J}{\partial \hat{Y}} \frac{1 - \sigma(A)}{\sigma(A)} \sigma(A)\sigma(A) \tag{24}$$

$$\frac{\partial J}{\partial A} = \frac{\partial J}{\partial \hat{Y}} \sigma(A)(1 - \sigma(A)) \tag{25}$$

Necesitamos la derivada parcial de la función de pérdida correspondiente a cada uno de los pesos. Pero como estamos recurriendo a la vectorización, podemos encontrarlo todo de una vez. Es por eso que hemos estado usando la notación mayúscula W en vez de $w_1, w_2, \ldots, w_n$.

$$\frac{\partial J}{\partial W} = \frac{\partial J}{\partial A} \frac{\partial A}{\partial W} \tag{26}$$

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial A} \frac{\partial A}{\partial b} \tag{27}$$

La derivación de los pesos queda partiendo de la ecuación 26:

$$\frac{\partial J}{\partial W} = \frac{\partial J}{\partial A} \frac{\partial}{\partial W}(W^T X + b) = \frac{\partial J}{\partial A} X \tag{28}$$

Y de la ecuación 27:

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial A} \frac{\partial}{\partial b}((W^T X + b) = \frac{\partial J}{\partial A} 1 = \frac{\partial J}{\partial A} \tag{29}$$

Se demostró desde un punto de vista matemático el concepto de *backpropagation* como se realiza la actualización de los pesos y sesgos utilizando el descenso por gradiente.

3.9. Descenso de gradiente estocástico (SGD)

Sin embargo el descenso de gradiente puede ser excepcionalmente lento en datasets muy grandes debido a que en cada iteración requiere calcular una predicción por cada punto de entrenamiento en nuestros datos de entrenamientos antes que actualizaciones nuestra matriz de pesos.

En cambio lo que se utiliza es una variante de éste, el descenso de gradiente estocástico o *Stochastic Gradient Descent (SGD)*. El SGD es una simple modificación del algoritmo de descenso de gradiente estándar que computa el gradiente y actualiza la matriz de pesos W en pequeños lotes o *batches* de datos de entrenamiento, en vez del *dataset* entero. Mientras esta modificación nos lleva a actualizaciones más ruidosas", también nos permite tomar más pasos a lo largo

del gradiente, llevando en ultima instancia a una convergencia más rápida y sin afectar negativamente a la pérdida y precisión del modelo.

En lugar de calcular nuestro gradiente en todo el conjunto de datos, en su lugar muestreamos nuestros datos, produciendo un lote. Evaluamos el gradiente en el lote y actualizamos nuestra matriz de peso W. Desde una perspectiva de implementación, también tratamos de aleatorizar nuestras muestras de entrenamiento antes de aplicar SGD ya que el algoritmo es sensible a los lotes.

En una implementación "purista"de SGD, el tamaño de su mini lote sería 1, lo que implica que muestrearíamos aleatoriamente un punto de datos del conjunto de entrenamiento, calcularíamos el gradiente, En una implementación "purista"de SGD, el tamaño de su mini lote sería 1, lo que implica que muestrearíamos aleatoriamente un punto de datos del conjunto de entrenamiento, calcularíamos el gradiente, y actualiza nuestros parámetros.

Sin embargo, a menudo utilizamos mini lotes que son>1. Los tamaños de lote típicos incluyen 32, 64, 128 y 256.

¿Por qué usar lotes de tamaños >1?

1. Ayudan a reducir la variación en la actualización de parámetros, lo que conduce a una convergencia más estable.

2. Las potencias de dos a menudo son deseables para los tamaños de lote, ya que permiten que las bibliotecas de optimización de álgebra lineal interna sean más eficientes.

En general, el tamaño del mini lote no es un hiperparámetro por el que debería preocuparse demasiado. Si está usando una GPU para entrenar su red neuronal, usted determina cuántos ejemplos de entrenamiento encajarán en su GPU y luego usa la potencia más cercana de dos, ya que el tamaño del lote se ajustará en la GPU. Para el entrenamiento de CPU, normalmente utiliza uno de los tamaños de lote enumerados anteriormente para asegurarse de cosechar los beneficios de las bibliotecas de optimización de álgebra lineal.

3.10. Sobreajuste y bajo-ajuste

El sobreajuste o *overfitting* y la falta de ajuste o *underfitting* [7] es muy importante para saber si el modelo predictivo está generalizando bien los datos o no. Un buen modelo debe poder generalizar bien los datos.

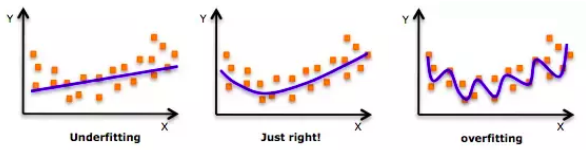


Figura 10: Distintas representaciones del ajuste en un mismo modelo.

En la Figura 10 **Izquierda** el modelo está sobreajustado, *i.e.* cuando funciona bien en el ejemplo de entrenamiento pero no funciona bien en datos no vistos. A menudo es el resultado de un modelo excesivamente complejo y ocurre porque el modelo está memorizando la relación entre el ejemplo de entrada (a

menudo llamado X) y la variable objetivo (a menudo llamada y) o, por lo tanto, no puede generalizar bien los datos. El modelo de sobreajuste predice el objetivo en el conjunto de datos de entrenamiento con mucha precisión.

En cambio en la Figura 10 **Derecha** se dice que el modelo predictivo tiene bajo-ajuste, si funciona mal en los datos de entrenamiento. Esto sucede porque el modelo no puede capturar la relación entre el ejemplo de entrada y la variable objetivo. Podría deberse a que el modelo es demasiado simple, es decir, las características de entrada no son lo suficientemente expresivas como para describir bien la variable objetivo. El modelo con bajo-ajuste no predice los objetivos en los conjuntos de datos de entrenamiento con mucha precisión.

Un buen modelo debe ser como el de la Figura 10 **Medio** que posee una buena precisión en su conjunto de datos de entrenamiento pero a su vez también tiene una buena *performance* con datos que no haya visto.

3.11. Regularización

Para disminuir los efectos del sobreajuste se utiliza la **regularización** que después de la tasa de aprendizaje, es el parámetro más importante de su modelo que puede ajustar.

Existen varios tipos de técnicas de regularización, como la regularización L1, la regularización L2 (comúnmente llamada "pérdida de peso") y Elastic Net, que se utilizan al actualizar la función de pérdida en sí, agregando un parámetro adicional para restringir la capacidad de el modelo.

La regularización nos ayuda a controlar la capacidad de nuestro modelo, asegurando que nuestros modelos sean mejores para hacer clasificaciones (correctas) en los puntos de datos en los que no fueron entrenados, lo que llamamos la capacidad de generalizar. Si no aplicamos la regularización, nuestros clasificadores pueden volverse demasiado complejos y ajustarse fácilmente a nuestros datos de entrenamiento, en cuyo caso perdemos la capacidad de generalizar a nuestros datos de prueba.

3.12. Los cuatro ingredientes de una red neuronal

Hay cuatro ingredientes principales que necesita para armar su propia red neuronal y algoritmo de aprendizaje profundo: un conjunto de datos, un modelo/arquitectura, una función de pérdida y una optimización.

3.12.1. Conjunto de datos

También llamado *dataset*, es el primer ingrediente en el entrenamiento de una red neuronal: los datos en sí mismos junto con el problema que estamos tratando de resolver definen nuestros objetivos finales.

La combinación de su conjunto de datos y el problema que está tratando de resolver influye en su elección en la función de pérdida, la arquitectura de red y el método de optimización utilizado para entrenar el modelo. Por lo general, tenemos pocas opciones en nuestro conjunto de datos (a menos que esté trabajando en un proyecto de pasatiempo): se nos da un conjunto de datos con cierta expectativa sobre cuáles deberían ser los resultados de nuestro proyecto. Depende de nosotros entrenar un modelo de aprendizaje automático en el conjunto de datos para que funcione bien en la tarea dada.

3.12.2. Función de pérdida

Dado nuestro conjunto de datos y objetivo objetivo, necesitamos definir una función de pérdida que se alinee con el problema que estamos tratando de resolver.

3.12.3. Modelo/Arquitectura

La arquitectura de su red puede considerarse la primera .[el]ecciónreal que tiene que hacer como ingrediente. Es probable que su conjunto de datos sea elegido para usted (o al menos ha decidido que desea trabajar con un conjunto de datos determinado). Y si está realizando una clasificación, probablemente utilizará la entropía cruzada como su función de pérdida. Sin embargo, su arquitectura de red puede variar dramáticamente, especialmente cuando con qué método de optimización elige entrenar su red.

3.12.4. Método de optimización

El ingrediente final es definir un método de optimización. El SGD se usa con bastante frecuencia. SGD sigue siendo el caballo de batalla del aprendizaje profundo: la mayoría de las redes neuronales se entrenan a través de SGD, aunque existen otros métodos de optimización como Adam. Luego debe establecer una tasa de aprendizaje adecuada, la fuerza de regularización y el número total de épocas para las que se debe entrenar la red.

4. Redes neuronales convolucionales

En las redes neuronales *feedforward* tradicionales, cada neurona en la capa de entrada está conectada a cada neurona de salida en la siguiente capa; a esto le llamamos una capa completamente conectada (FC). Sin embargo, en las CNN, no utilizamos capas FC hasta las últimas capas de la red. De este modo, podemos definir una CNN como una red neuronal que se intercambia en una capa especializada çonvolucional.[en] lugar de una capa çompletamente conectada"para al menos una de las capas de la red.

Luego se aplica una función de activación no lineal, como ReLU, a la salida de estas convoluciones y el proceso de convolución =>la activación continúa (junto con una mezcla de otros tipos de capas para ayudar a reducir el ancho y la altura del volumen de entrada y ayudar a reducir sobreajuste) hasta que finalmente lleguemos al final de la red y apliquemos una o dos capas FC donde podamos obtener nuestras clasificaciones finales de salida.

Cada capa en una CNN aplica un conjunto diferente de filtros, típicamente cientos o miles de ellos, y combina los resultados, alimentando la salida a la siguiente capa en la red. Durante el entrenamiento, una CNN aprende automáticamente los valores de estos filtros.

La última capa en una CNN usa estas características de nivel superior para hacer predicciones sobre el contenido de la imagen. En la práctica, las CNN nos brindan dos beneficios clave: invariancia local y composicionalidad.

1. El concepto de invariancia local nos permite clasificar una imagen como que contiene un objeto en particular, independientemente de en qué parte

de la imagen aparezca el objeto. Obtenemos esta invariancia local mediante el uso de çapas de agrupación"que identifica regiones de nuestro volumen de entrada con una alta respuesta a un filtro particular.

2. Composicionalidad. Cada filtro compone un parche local de *features* de nivel inferior en una representación de nivel superior, similar a cómo podemos componer un conjunto de funciones matemáticas que se basan en la salida de funciones anteriores: $f(g(x(h(x)))$ - esta composición permite que nuestra red aprenda *features* más ricas más profundamente en la red. El concepto de crear *features* de nivel superior a partir de las de nivel inferior es exactamente la razón por la cual las CNN son tan poderosas en visión por computadora.

Referencias

[1] David I Poole, Randy G Goebel, and Alan K Mackworth. *Computational intelligence*. Oxford University Press New York, 1998.

[2] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[3] URL `https://origin-symwisedownload.symantec.com/resources/webguides/contentanalysis/24/Content/Topics/Concepts/aml_process.htm`.

[4] Tariq Rashid. *Make your own neural network*. CreateSpace Independent Publishing Platform, 2016.

[5] Adrian Rosebrock. *Deep Learning for Computer Vision with Python: Starter Bundle*. PyImageSearch, 2017.

[6] freeCodeCamp.org. Demystifying gradient descent and backpropagation via logistic regression based image..., Jul 2018. URL `https://www.freecodecamp.org/news/demystifying-gradient-descent-and-backpropagation-via-logistic-regression-based-image-`

[7] What are the key trade-offs between overfitting and underfitting? URL `https://www.quora.com/What-are-the-key-trade-offs-between-overfitting-and-underfitting`.