Números pseudoaleatorios y Redes Neuronales

Paula Rosado Fernández

Análisis Numérico
Universidad Complutense de Madrid

23/06/2021

Índice general

Capítulo 1

Simulando números aleatorios de una distribución uniforme

Un ordenador digital no puede generar números aleatorios, y generalmente no es conveniente conectarlo a una fuente externa de eventos aleatorios. Para la mayoría de aplicaciones en estadística esto no supone un problema si existe una fuente de números pseudoaleatorios. Es decir, muestras que parecen haber sido generadas por una distribución conocida.

1.1. Generador lineal congruencial

D.G Lehmer en 1948[1] propuso el *generador lineal congruencial* como una fuente de números aleatorios. En este generador, cada número determina su sucesor. Tiene la siguiente forma:

$$x_i \equiv (ax_{i-1} + c) \pmod{m} \qquad 0 \leq x_i < m \tag{1.1}$$

Dónde a es el 'multiplicador', c el 'incremento' y m el 'módulo' del generador. Normalmente c en (1.1) suele escogerse como 0, y en este caso al generador se le llama 'generador congruencial multiplicativo':

$$x_i \equiv ax_{i-1} \pmod{m} \qquad 0 < x_i < m \tag{1.2}$$

El valor inicial de la recursión, x_0, es la "semilla". Cada x_i se escala al intervalo unitario (0,1), dividiéndolo por m.

$$u_i = x_i/m \tag{1.3}$$

Si la a y la m se escogen apropiadamente, las $u_i's$ parecerán aleatorias y uniformemente distribuidas entre 0 y 1. La recurrencia en (1.2) para los enteros es equivalente a la recurrencia:

$$u_i \equiv ux_{i-1} \quad (\text{mód } 1) \qquad 0 < u_i < 1 \tag{1.4}$$

Esta recurrencia (1.4) tiene una interesante relación con el modelo autoregresivo lineal de primer orden

$$U_i = \rho u_{i-1} + E_i \tag{1.5}$$

Dónde ρ es el coeficiente de autoregresión y E_i es una variable aleatoria con una distribución uniforme entre 0 y 1.

Porque x_i se determina a partir de x_{i-1} y como solo hay m posibles valores diferentes y a parte $x_{i-1} = 0$ no está permitido que esté en un generador multiplicativo, el periodo máximo del generador lineal congruencial es m - 1.

La elección de las constantes 'a', 'c' y el módulo 'm' es la que determinará la calidad de los números pseudoaleatorios. Se ha de resaltar que la elección de un m de gran tamaño no asegurará que el periodo sea el valor de m-1 (Figura 1.2).

Actualmente los números usados como módulo son los primos de Marsenne[2] ($2^p - 1$p es un número primo). Para conseguir que el periodo sea m -1, 'm' debe ser primo y 'a' una raiz primitiva modulo m. Los valores más utilizados en la literatura son m = $2^{31} - 1$ y a = 7^5.

1.1.1. Ejecución del algoritmo

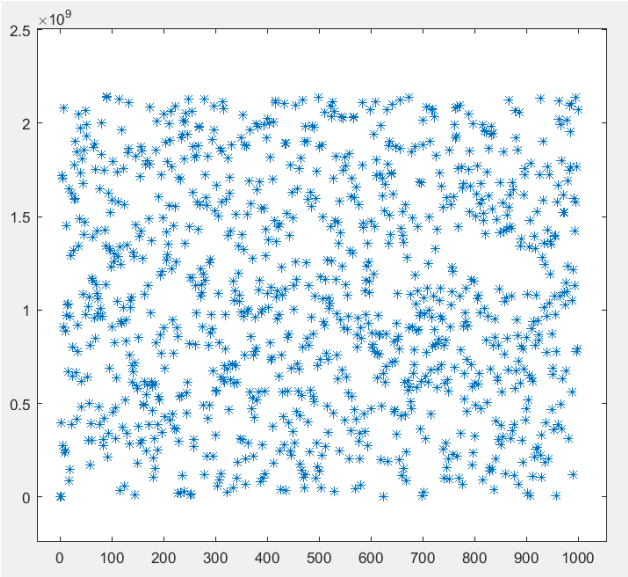

Figura 1.1: $m = 2^{31} - 1$ *y* $a = 7^5$

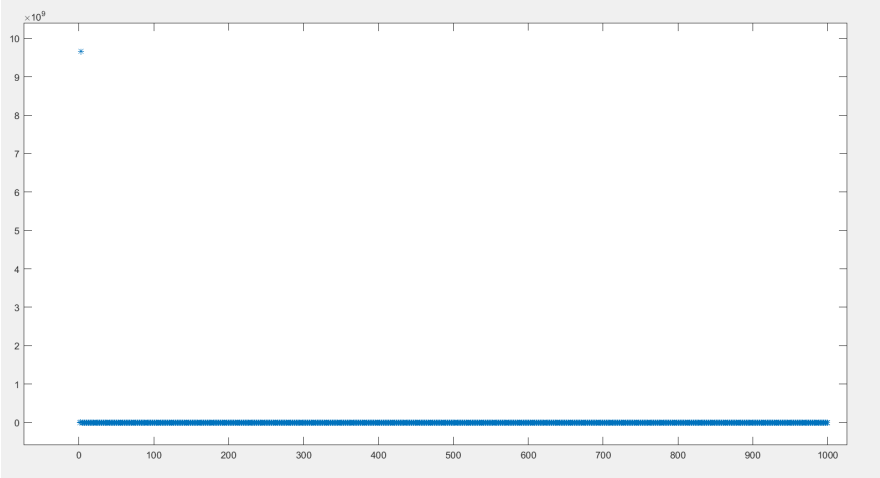

Figura 1.2: $m = 2^{40}$ *y* $a = 8^3$

1.2. Generador no lineal congruencial: Blum Blum Shub

Este generador fue creado por Lenore Blum, Manuel Blum y Michael Shub en 1968.[3] Tiene la siguiente forma:

$$x_i \equiv x_{i-1}^2 \pmod{M} \tag{1.6}$$

Donde x_0 es una 'semilla' aleatoria, y $M = p * q$ siendo p y q números primos distintos y congruentes a 3 módulo 4. Blum Blum y Shub demostraron que al tener esta forma, la salida del generador no es predecible en tiempo polinomial sin saber los valores de p y q.

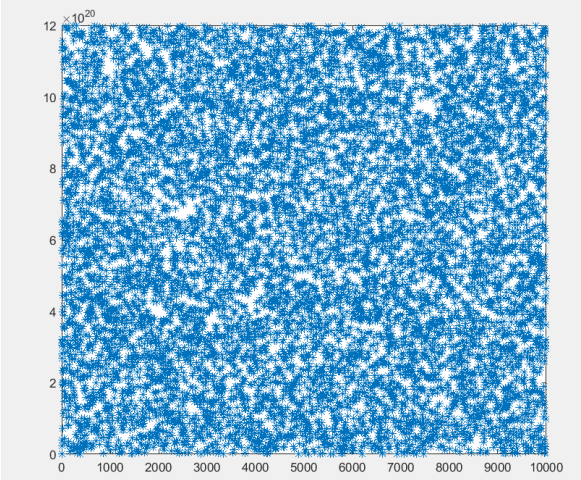

Figura 1.3: *p = 30000000091, q = 40000000003*[4]

Capítulo 2

Métodos para generar distribuciones no uniformes

2.1. Método de la Transformada inversa

Este método se basa en el siguiente lema: Supongamos U $\sim$ Unif [0,1] (variable aleatoria distribuida unifórmemente) y F una función de distribución acumulada (FDA) de una variable aleatoria X. Entonces:

$$X = F^{-1}(U).^{5} \tag{2.1}$$

La demostración es de la siguiente manera:

$$P(X \leq x) = P(F^{-1}(U) \leq x) = P(U \leq F(x)) = F(x). \tag{2.2}$$

El algoritmo para distribuciones continuas es el siguiente:

1. Generar U $\sim$ Unif [0,1]
2. Asignamos $X = F^{-1}(U)$

Este método funciona muy bien cuando la inversa de la función de distribución es fácil de calcular. Esto no siempre ocurre, por ejemplo para una variable aleatoria que sigue una distribución exponencial este método sería el idóneo, en cambio para una que siguiera una distribución normal el método se convertiría en un proceso difícil de realizar.

2.1.1. Ejemplo del algoritmo

En este ejemplo generaremos números aleatorios que siguen una distribución exponencial. Para ello utilizaremos su función de distribución:

$$F(x) = 1 - e^{-\lambda x} \qquad x \geq 0 \tag{2.3}$$

La función inversa de F(x) es la siguiente:

$$F^{-1}(y) = -\frac{\ln 1 - y}{\lambda} \tag{2.4}$$

Para la simulación realizada en matlab el valor λ es 3 y para comprobar que los números aleatorios generados siguen una distribución exponencial, se han representado las frecuencias de las apariciones en forma de histograma.

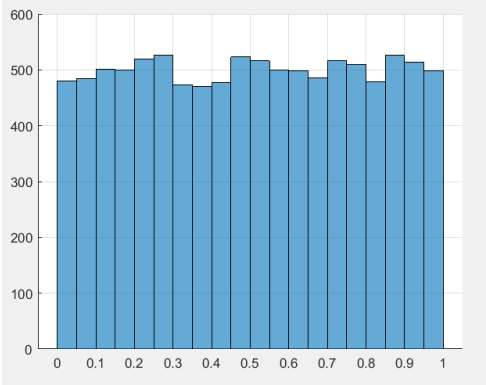

Figura 2.1: *Histograma de la muestra de la distribución uniforme [0,1]*

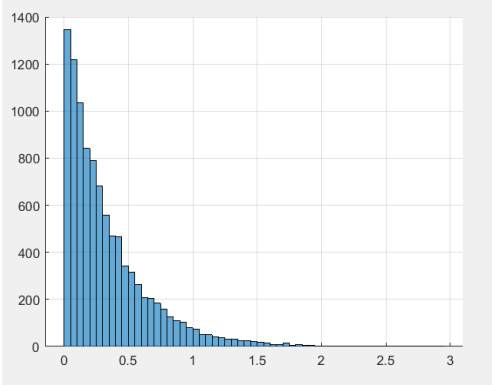

Figura 2.2: *Histograma de los números aleatorios generados con una distribución exponencial*

2.2. Método de aceptación/rechazo

La idea principal de este método es la siguiente:

Supongamos que se tiene una distribución objetivo $\pi(x)$, de la que se quiere obtener una muestra de números aleatorios, pero va a ser difícil generar los números directamente de ella. El Método de aceptación/rechazo dice que siempre que haya una distribución conocida g(x) (de la cual sea fácil hacer un muestreo) y $\pi(x) \leq cg(x)$, siendo c una constante, entonces podemos utilizar una muestra aleatoria de cg(x) para obtener los números aleatorios de $\pi(x)$.

De este modo la función cg(x) 'envolvería' todos los puntos del soporte de X de la distribución $\pi(x)$ y a continuación se realizaría un muestreo de puntos aleatorios. De estos se aceptarán si están debajo de la función $\pi(x)$ y se rechazarán el resto.

El algoritmo es el siguiente:[6]

1. Generar X de la distribución con función de densidad g(x)

2. Generar u $\sim$ Unif [0,1]

3. Si u $\leq \frac{\pi(X)}{cg(X)}$ aceptamos X

3.1 Si no volvemos al paso 1

2.2.1. Ejemplo del algoritmo

Para este ejemplo he tomado una función continua creciente:

$$\pi(x) = \frac{3}{2}x^3 + \frac{12}{5}x^2 - \frac{1}{6}x + \frac{2}{5} \qquad 0 \leq x \leq 1 \tag{2.5}$$

Como distribución con función de densidad g(x) he tomado una uniforme [0,1] y para escoger un valor de c adecuado he evaluado $\pi(x)$ en el extremo del intervalo. Teniendo que:

$$c = \pi(6) = 2{,}77095. \tag{2.6}$$

La gráfica de $\pi(x)$ y la distribución uniforme continua sería el siguiente:

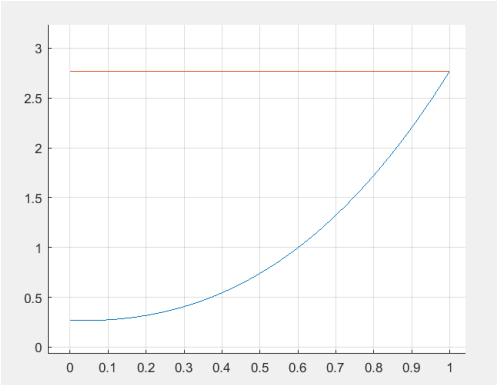

Figura 2.3: *Curva azul: Función de densidad* $\frac{120}{179}(\frac{3}{2}x^3 + \frac{12}{5}x^2 - \frac{1}{6}x + \frac{2}{5})$ *,recta naranja: distribución unifirme*

Tras aplicar el algorimo para una muestra aleatoria de 10000 números y queriendo obtener 100 números aleatorios distribuidos según $\pi(x)$, obtenemos el siguiente histograma:

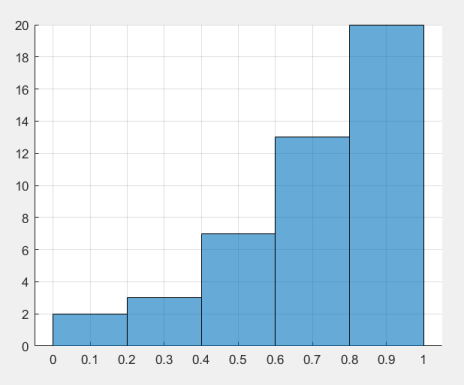

Figura 2.4:

Capítulo 3

Aprendizaje supervisado: Redes Neuronales

3.1. Definiciones

Los algoritmos de aprendizaje supervisado basan su aprendizaje en un conjunto de datos de entrenamiento previamente etiquetados. Por etiquetado se entiende que para cada ocurrencia del conjunto de datos de entrenamiento conocemos el valor de su atributo objetivo. Esto le permitirá al algoritmo poder "aprender" una función capaz de predecir el atributo objetivo para un conjunto de datos nuevo.
Existen dos técnica sde aprendizaje supervisado:

Clasificación: cuando el atributo objetivo es una clase, entre un número limitado de clases. Con clases se entiende como categorías arbitrarias según el tipo de problema.

Regresión: cuando el atributo objetivo es un número. Es decir, el resultado será un valor numérico, dentro de un conjunto infinito de posibles resultados.

La red neuronal es un algoritmo que pertenece a este tipo de aprendizaje y se puede emplear para problemas tanto de clasificación como regresión, en el ejemplo que se desarrollará posteriormente se utiliza para resolver un problema de clasificación.

3.2. Red neuronal

Para comprender mejor el funcionamiento de una red neuronal, se decompondrá el algoritmo en varios pasos. Pero antes de esto, se explicará la estructura de una red neuronal y los elementos que la componen.

3.2.1. Estructura y elementos

Para entender mejor la idea este algoritmo, se utilizará como ejemplo el problema que se ha implementado de clasificación. El problema se basa en determinar que número es, una imagen de un número escrito a mano. La red neuronal recibe 784 inputs, que representan los pixeles de una imagen de 28x28, cada pixel tiene un valor entre 0 y 1 representando la escala de gris, siendo 1 negro y 0 blanco. La salida de la red neuronal compone de diez valores entre 0-9.

La red neuronal se compone de capas y cada capa de neuronas. Existen tres tipos de capas:
Capa de entrada: está compuesta por neuronas que reciben datos.
Capa de salida: se compone de neuronas que proporcionan la respuesta de la red neuronal.
Capa oculta: no tiene una conexión directa con el entorno.

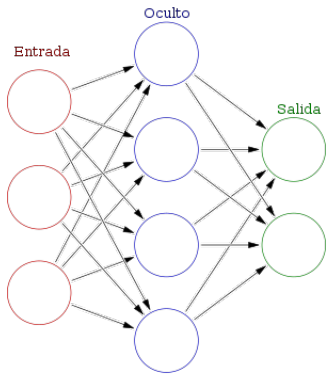

Figura 3.1: *Red neuronal de tres capas*

Denotaremos x como la entrada de red neuronal, e $y = y(x)$ como la salida deseada. Para este problema y se representa como un vector de 10 dimensiones. Por ejemplo si la salida esperada es un 3 entonces, $y(x) = (0001000000)^T$ siendo T la traspuesta.

La red neuronal funciona de tal manera que se establecen aleatoriamente unos 'pesos' y 'umbrales', estos se utilizan en una combinación lineal con las entradas y mediante una serie de operaciones se calcula la salida de la red neuronal $\hat{y}$.

El algoritmo permite encontrar pesos y umbrales de tal manera que $\hat{y}$ se aproxime a y para todas las entradas x. Para cuantificar el progreso de este objetivo se define una función de coste o pérdida.

$$C(w,b) = \frac{1}{2n}\sum_{i=1}^{n} ||y_i(x) - \hat{y}_i||^2 \tag{3.1}$$

Dónde w es el peso, b el umbral, n el número total de entradas. Se puede observar que $C(w,b)$ es positiva, pues la suma de los cuadrados nunca podrá ser negativa y $C(w,b)$ se hace pequeña ($C(w,b) \sim 0$) cuando y es aproximadamente igual a $\hat{y}$, por lo tanto el objetivo es minimizar la función de coste y para ello utilizaremos el *método del descenso del gradiente*.

3.2.2. Mecanismos que se emplean en la programación de la red neuronal

[7] Antes de adentrarnos con la explicación del *método del descenso del gradiente* y que algoritmo se emplea para programarlo, enseñaremos cómo se calcula $\hat{y}$.

A través de las ecuaciones (3.2) y (3.3) se puede explicar como se realiza este cálculo:

$$a_j^1 = x \in \mathbb{R}^{n_1} \tag{3.2}$$

$$a_j^l = \sigma(\sum_k w_{jk}^l a_k^{l-1} + b_j^l) \in \mathbb{R}^{n_l} \qquad para\ l = 2, 3, 4..L \tag{3.3}$$

Las capas se representan como l, a denota para la 'activación' de la neurona, el valor de cada neurona, para la capa de entrada es la propia entrada (x), y para todas las capas exceptuando esta, a se define como la ecuación 3.3. Dónde a_j^l representa el valor de activación j-ésima neurona de la capa l-ésima, w_{jk}^l el peso de la conexión entre k-ésima neurona de las (l-1)-ésima capa con la j-ésima neurona en la capa l, a_k^{l-1} la activación de la k-ésima neurona en la capa l-ésima y b_j^l el umbral j-éismo en al capa l-ésima. Siendo k el núemero de neuronas en la capa (l-1)-ésima. Finalmente σ representa una función no lineal, a esta función se le llama la *función de activación*, para este problema se ha elegido la sigmoidal(3.4)

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3.4}$$

La elección de esta función reside en primer lugar en el tipo de salida del problema (un valor entre 0 y 1), el rango de esta función se encuentra entre 0 y 1. Además su derivada presenta una forma sencilla de computar[8] $\sigma'(x) = \sigma(x)(1 - \sigma(x))$.

Para reescribir esta expresión en forma matricial, se define una *matriz de pesos* w^l para cada capa, donde la entrada en la j-ésima fila y la k-ésima columna es w_{jk}^l. De manera similar se definen el vector de umbrales b^l y el de activaciones a^{l-1}. Aplicar σ a un vector es simplemente aplicar σ a cada elemento del vector $\sigma(v)_j = \sigma(v_j)$. De esta manera podemos reescribir la ecuación 3.3:

$$a^l = \sigma(W^l a^{l-1} + b^l) \in \mathbb{R}^{n_l} \qquad para\ l = 2, 3, 4..L \tag{3.5}$$

Cuando l es la última capa a a^l se le denomina $\hat{y}$.

Método del descenso del gradiente

Intuición del descenso del gradiente:

Imaginemos que tenemos una función que tiene una entrada y una salida. ¿Cómo encontramos un valor de entrad que minimice el valor de la función? Un posible algortimo sería el siguiente:

Aleatoriamente establecemos un punto en la función calculamos su pendiente si es negativa nos movemos a la derecha y es positiva a la izquierda, hasta encontrar un punto donde la

pendiente sea cercana a 0, entonces sabremos que habremos alcanzado un local mínimo de la función.

El descenso del gradiente representa esta idea, solo que en más dimensiones. Por ejemplo si estuvieramos en 2D en vez de calcular la pendiente tendríamos que calcular la dirección. El gradiente nos proporciona la dirección para incrementar la función lo más rápido posible, por lo tanto el '- gradiente' nos proporcionará la dirección para decrementar la función lo más rápido posible.
Quedándonos con el ejemplo de 2D, imaginemos que queremos minimizar una función $C(v)$, esta función tiene dos variables v_1 y v_2 y queremos encontrar en mínimo local de la función. Si una pelota se mueve una poco Δv_1 en la dirección v_1 y otro poco Δv_2 en la dirección v_2, el cálculo nos dice que C cambia de la siguiente manera:

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2 \tag{3.6}$$

Necesitamos encontrar la manera de elegir Δv_1 y Δv_2 para hacer ΔC. Definimos Δv como el vector de cambios en v, $\Delta v \equiv (\Delta v_1, \Delta v_2)^T$ y el gradiente de C, como el vector de las derivadas parciales $\nabla C \equiv (\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2})^T$. Con estas definiciones ΔC se puede reescribir como:

$$\Delta C \approx \nabla C \Delta v \tag{3.7}$$

Eligiendo el Δv adecuado podemos hacer que ΔC sea negativo, en particular si elegimos:

$$\Delta v = -\eta \nabla C \tag{3.8}$$

Siendo η la tasa de aprendizaje. La ecuación 3.7 nos dice que $\Delta C \approx -\nabla C \eta \nabla C = -\eta ||\nabla C||^2$. Como $||\nabla C||^2 \geq 0$, esto nos garantiza que $\Delta C \leq 0$. De esta forma, utilizando la ecuación 3.8 podemos definir cuanto se debe mover la pelota (en este ejemplo) para llegar a un mínimo local.

$$v \rightarrow v^{'} = v - \eta \nabla C \tag{3.9}$$

Igual que hemos calculado descenso del gradiente para un vector de dos variables, se puede hacer de más. Aplicando esta técnica a nuestro problema, aplicaremos el descenso del gradiente para encongtrar los pesos y umbrales que minimicen la función de coste 3.1. De esta manera:

$$w_k \rightarrow w_k^{'} = w_k - \eta \frac{\partial C}{\partial w_k} \tag{3.10}$$

$$b_l \rightarrow b_l^{'} = b_l - \eta \frac{\partial C}{\partial b_l} \tag{3.11}$$

Antes de adentrarnos en como programar el descenso del gradiente es importante puntualizar un problema que nos encontraremos en la práctica. La función de coste tiene la forma $C = \frac{1}{N}\sum_{i=1}^{N} C_{x^i}$. En la práctica calcular el gradiente ∇C tenemos que calcular primero el gradiente ∇C_x y luego hacer la media, quedando $\nabla C = \frac{1}{N}\sum_{i=1}^{N} \nabla C_{x^i}$. Desafortunadamente cuando el número de inputs es muy grande, este cálculo es demasiado lento.

De este modo, introducimos el *descenso del gradiente estocástico*. La idea es estimar el gradiente ∇C calculando ∇C_x para un conjunto de aleatorio de inputs. Haciendo la media de estos pequeños conjuntos, podemos encontrar una estimación rápida y cercana del verdadero gradiente ∇C y esto ayuda a acelerar el descenso del gradiente y por lo tanto el aprendizaje de la red.
Para aclarar un poco cómo funciona, el descenso del gradiente estocástico funciona cogiendo un número pequeño de m inputs (elegidos aleatoriamente). Llamaremos a estos inputs $X_1, X_2, ..., X_m$ *mini-batch*. De esta manera:

$$\frac{\sum_{j=1}^{m} \nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C \tag{3.12}$$

Por lo tanto las ecuaciones 3.10 y 3.11 se reescribirian de la siguiente manera:

$$w_k \to w_k^{'} = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial w_k} \tag{3.13}$$

$$b_l \to b_l^{'} = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial b_l} \tag{3.14}$$

Una vez ya comprendido qué es el método del descenso del gradiente y la utilización del método estocástico, pasaremos a explicar cómo se programa y qué algoritmo se emplea para ello. Es en este contexto dónde se introduce el *algoritmo de Backpropagation*:

1. **Input** x: se establece la correspondiente activación a^1 para la capa de entrada
2. **Feedforward:** Para cada capa $l = 2, 3, .., L$ se calcula $z^l = w^l a^{l-1} + b^l$ y $a^l = \sigma(z^l)$
3. **Output error** δ^L: se calculan los vectores $\delta^L = \nabla_a C \odot \sigma^{'}(z^L)$
4. **Backpropagate error**: Para cada $l = L-1, L-2, ..., 2$ se calcula
$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma^{'}(z^l)$
5.**Output:** El gradiente de la función de coste es: $\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l$ y $\frac{\partial C}{\partial b_j^l} = \delta_j^l$

Ahora se parará a explicar todas las ecuaciones involucradas, son cuatro las ecuaciones fundamentales de este algoritmo:[9]

Ecuación para el error de la capa de salida δ^L

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma^{'}(z_j^L) \tag{3.15}$$

El primer término empezando por la derecha mide cómo de rápido cambia el coste como función de la j-ésima activación. El segundo término, mide cómo de rápido la función de activación cambia en z_j^L. Esta ecuación es fácil de calcular pues z_j^L se obtiene en el proceso del cáculo de $\hat{y}$ y la derivada de la función de activación requiere un coste adicional muy pequeño. La forma exacta de $\frac{\partial C}{\partial a_j^L}$ dependrá de la forma de la función de coste. No obstante, si la esta última es conocida, no debería ser difícil calcularla. Como para este ejemplo se

está usando la función de coste cuadrática $C = \frac{1}{2}\sum_j (y_j - a_j^L)^2$ entonces $\frac{\partial C}{\partial a_j^L} = (a_j^L - y_j)$. Para poder programarla necesitamos que la ecuación 3.15 esté en forma de matriz, siendo esta:

$$\delta_j^L = (a_j^L - y_j) \odot \sigma'(z_j^L) \tag{3.16}$$

Ecuación para el error δ^l en términos del error en la siguiente capa δ^{l+1}

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{3.17}$$

Para comprender mejor esta ecuación supongamos que sabemos el error δ^{l+1} en la capa (l+1)-ésima. Cuando aplicamos la traspuesta de la matriz de pesos, se puede pensar intuitivamente como si moviesemos el error hacia atrás en la red, dándonos una clase de medida del error en la salida de la l-ésima capa. Después aplicamos el producto Haddamard, este mueve el error hacia atrás através de la función de activación en la capa l, dándo el error δ^l en la capa l.
Combinando las ecuaciones 3.16 y 3.17 podemos calcular el error para cualquier capa en la red, primero aplicando la 3.16 y luego aplicando la 3.17 repetidamente.

Ecuación para la tasa de cambio del coste respecto a cualquier umbral de la red

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{3.18}$$

De modo que el error δ_j^l es exactamente igual a la tasa de cambio $\frac{\partial C}{\partial b_j^l}$

Ecuación para la tasa de cambio del coste respecto a cualquier peso de la red

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l \tag{3.19}$$

De esta manera sabemos cómo calcular la derivada parcial en término de los valores de error en la misma capa y la activación en la anterior.

Una vez entendido cómo se programa del descenso del gradiente, utilizando el algoritmo de backpropagation, se desarrollará cómo cambia el algoritmo al realizarlo con el descenso del gradiente estocástico:

1. **Establecer un conjunto de entrada**
2. **Para cada conjunto de datos** x: Establecer la activación para la primera capa $a^{x,1}$ y seguir los siguientes pasos

Feedforward: Para cada capa $l = 2, 3, .., L$ se calcula $z^{x,l} = w^l a^{x,l-1} + b^l$ y $a^l = \sigma(z^{x,l})$

Output error δ^L: se calculan los vectores $\delta^{x,L} = \nabla_a C_x \odot \sigma'(z^{x,L})$

Backpropagate error: Para cada $l = L-1, L-2, ..., 2$ se calcula
$\delta^{x,l} = ((w^{l+1})^T \delta^{x,l+1}) \odot \sigma'(z^{x,l})$

3. **Descenso del gradiente:** Para cada $l = L, L-1, .., 2$ actualizar los pesos de acuerdo con la regla: $w_k \to w_k' = w_k - \frac{\eta}{m}\sum_x \delta^{x,l}(a^{x,l-1})^T$ y los umbrales $b_l \to b_l' = b_l - \frac{\eta}{m}\sum_x \delta^{x,l}$

Cuando hemos ejecutado todos lo minibatches se dice que se ha completado una *época*, se pueden ejecutar tantas épocas como se deseen.

3.3. Ejemplo de red neuronal

Cómo se ha comentado anteriormente, se ha programado una red neuronal desde cero, para resolver un problema de clasificación. Este problema recibe una serie de imágenes de números (del cero al nueve) escritos a mano y la red neuronal debe determinar que número es. El dataset (llamado MNIST[10]) está compuesto por 60000 imágenes, 50000 de ellas se emplearán para entrenar la red y 10000 para el test. La entrada de la red está compuesta por 784 neuronas, puesto que la imagen se representa como un vector de 784 valores, representando los píxeles de la imagen de 28x28. Cada componente del vector es un número real entre 0 y 1, 0 representa el blanco y 1 el negro, todo número entre ellos representa la tonalidad de gris de la imagen. La capa de salida de la red está compuesta por diez neuronas puesto que pueden ser diez posibles valores (0-9).

Para evaluar el rendimiento de la red se ha creado un método que mide por cada época cuántas imágenes ha acertado la red respecto al conjunto de evalución/test. Esta evaluación se puede realizar gracias a que cada neurona de la salida de la red, representa el propio número de la imagen. Por lo tanto, la posición de la neurona con mayor valor de activación (una vez entrenada) representará el número que clasifica la red. De este modo se puede ver si la posición coincide con el número que representa la imagen (este número está en el data set, junto al vector de píxeles).

A continuación se mostrará los resultados, para distintos hiperparámetros:
Para una red con 100 neuronas en la capa oculta, una tasa de aprendizaje de 0.001, 10 minibatches y 30 épocas.

```
net =Network([784, 100, 10])
net.SGD(training_data2, 30, 10, 0.001, test_data)
```

```
Epoch 0 : 1389 / 10000
Epoch 1 : 1414 / 10000
Epoch 2 : 1543 / 10000
Epoch 3 : 1634 / 10000
Epoch 4 : 1710 / 10000
Epoch 5 : 1778 / 10000
Epoch 6 : 1846 / 10000
Epoch 7 : 1894 / 10000
Epoch 8 : 1927 / 10000
Epoch 9 : 1958 / 10000
Epoch 10 : 1974 / 10000
Epoch 11 : 2010 / 10000
Epoch 12 : 2030 / 10000
Epoch 13 : 2051 / 10000
Epoch 14 : 2077 / 10000
Epoch 15 : 2110 / 10000
Epoch 16 : 2137 / 10000
Epoch 17 : 2163 / 10000
Epoch 18 : 2184 / 10000
Epoch 19 : 2195 / 10000
Epoch 20 : 2209 / 10000
Epoch 21 : 2225 / 10000
Epoch 22 : 2230 / 10000
Epoch 23 : 2240 / 10000
Epoch 24 : 2263 / 10000
Epoch 25 : 2279 / 10000
Epoch 26 : 2296 / 10000
Epoch 27 : 2310 / 10000
Epoch 28 : 2321 / 10000
Epoch 29 : 2336 / 10000
```

Figura 3.2:

Para una red con 100 neuronas en la capa oculta, una tasa de aprendizaje de 100.0, 10 minibatches y 30 épocas.

```
net2 = Network([784, 30, 10])
net2.SGD(training_data2, 30, 10, 100.0,test_data)
```

```
Epoch 0 : 1032 / 10000
Epoch 1 : 1013 / 10000
Epoch 2 : 1013 / 10000
Epoch 3 : 1013 / 10000
Epoch 4 : 1013 / 10000
Epoch 5 : 1013 / 10000
Epoch 6 : 1013 / 10000
Epoch 7 : 1013 / 10000
Epoch 8 : 1013 / 10000
Epoch 9 : 1013 / 10000
Epoch 10 : 1013 / 10000
Epoch 11 : 1013 / 10000
Epoch 12 : 1013 / 10000
Epoch 13 : 1013 / 10000
Epoch 14 : 1013 / 10000
Epoch 15 : 1013 / 10000
Epoch 16 : 1013 / 10000
Epoch 17 : 1013 / 10000
Epoch 18 : 1013 / 10000
Epoch 19 : 1013 / 10000
Epoch 20 : 1013 / 10000
Epoch 21 : 1013 / 10000
Epoch 22 : 1013 / 10000
Epoch 23 : 1013 / 10000
Epoch 24 : 1013 / 10000
Epoch 25 : 1013 / 10000
Epoch 26 : 1013 / 10000
Epoch 27 : 1013 / 10000
Epoch 28 : 1013 / 10000
Epoch 29 : 1013 / 10000
```

Figura 3.3:

Para una red con 100 neuronas en la capa oculta, una tasa de aprendizaje de 3.0, 10 minibatches y 30 épocas.

```
net3 =Network([784, 100, 10])
net3.SGD(training_data2, 30, 10, 3.0,test_data)
```

```
Epoch 0 : 6510 / 10000
Epoch 1 : 6638 / 10000
Epoch 2 : 6708 / 10000
Epoch 3 : 6732 / 10000
Epoch 4 : 6717 / 10000
Epoch 5 : 6770 / 10000
Epoch 6 : 6764 / 10000
Epoch 7 : 6814 / 10000
Epoch 8 : 6837 / 10000
Epoch 9 : 6853 / 10000
Epoch 10 : 6868 / 10000
Epoch 11 : 6857 / 10000
Epoch 12 : 6912 / 10000
Epoch 13 : 6959 / 10000
Epoch 14 : 6987 / 10000
Epoch 15 : 6986 / 10000
Epoch 16 : 7033 / 10000
Epoch 17 : 7175 / 10000
Epoch 18 : 7760 / 10000
Epoch 19 : 7765 / 10000
Epoch 20 : 7781 / 10000
Epoch 21 : 7764 / 10000
Epoch 22 : 7801 / 10000
Epoch 23 : 7799 / 10000
Epoch 24 : 7790 / 10000
Epoch 25 : 7821 / 10000
Epoch 26 : 7847 / 10000
Epoch 27 : 7844 / 10000
Epoch 28 : 7858 / 10000
Epoch 29 : 7837 / 10000
```

Figura 3.4:

Para una red con 30 neuronas en la capa oculta, una tasa de aprendizaje de 3.0, 10 minibatches y 30 épocas.

```
net3 =Network([784, 30, 10])
net3.SGD(training_data2, 30, 10, 3.0,test_data)
```

```
Epoch 0 : 9129 / 10000
Epoch 1 : 9205 / 10000
Epoch 2 : 9257 / 10000
Epoch 3 : 9311 / 10000
Epoch 4 : 9354 / 10000
Epoch 5 : 9340 / 10000
Epoch 6 : 9377 / 10000
Epoch 7 : 9393 / 10000
Epoch 8 : 9411 / 10000
Epoch 9 : 9431 / 10000
Epoch 10 : 9415 / 10000
Epoch 11 : 9428 / 10000
Epoch 12 : 9450 / 10000
Epoch 13 : 9452 / 10000
Epoch 14 : 9419 / 10000
Epoch 15 : 9461 / 10000
Epoch 16 : 9462 / 10000
Epoch 17 : 9446 / 10000
Epoch 18 : 9476 / 10000
Epoch 19 : 9458 / 10000
Epoch 20 : 9486 / 10000
Epoch 21 : 9473 / 10000
Epoch 22 : 9462 / 10000
Epoch 23 : 9475 / 10000
Epoch 24 : 9465 / 10000
Epoch 25 : 9454 / 10000
Epoch 26 : 9442 / 10000
Epoch 27 : 9462 / 10000
Epoch 28 : 9482 / 10000
Epoch 29 : 9469 / 10000
```

Figura 3.5:

3.3.1. Discusión sobre los resultados

Se puede observar que los mejores resultados aparecen cuando la tasa de aprendizaje no es ni excesivamente elevada, ni excesivamente baja, además podemos ver un incremento de la calidad de los resultados cuando decrementamos neuronas a la capa oculta (3.5).
Cuando la tasa de aprendizaje es baja (figura 3.2) la mayor tasa de aciertos es del $23,36\,\%$, aunque sea un porcenjate bajo, se puede observar como va mejorando a lo largo de las épocas. En cambio cuando la tasa de aprendizaje es demasiado elevada (3.3) no se observa que haya ningún incremento de la calidad del aprendizaje de la red.

3.3.2. Ejemplos de aciertos y fallos de la red

Para tener una idea más visual de las imágenes que la red neuronal clasifica bien y no, se han implementados dos funciones que muestran que imágenes clasifica de manera correcta y cuales no.

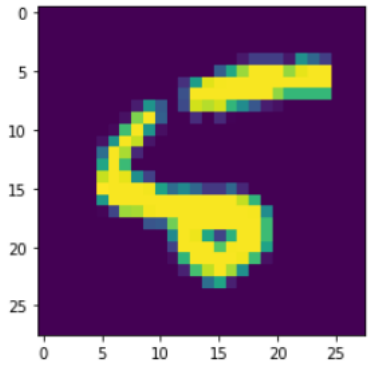

Figura 3.6: *Imagen que clasifica cómo un 6, cuando representa un 5*

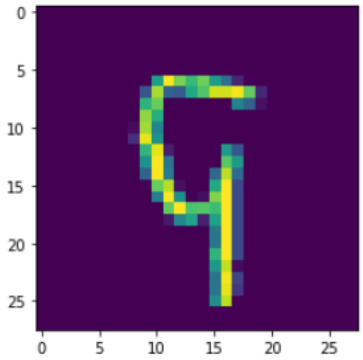

Figura 3.7: *Imagen que clasifica cómo un 4, cuando representa un 9*

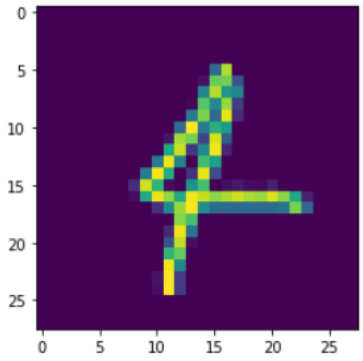

Figura 3.8: *Imagen que clasifica cómo un 1, cuando representa un 4*

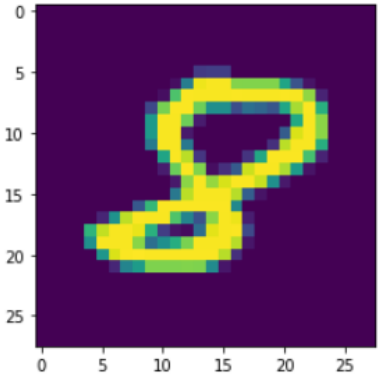

Figura 3.9: *Imagen que clasifica cómo un 8, y es un 8*

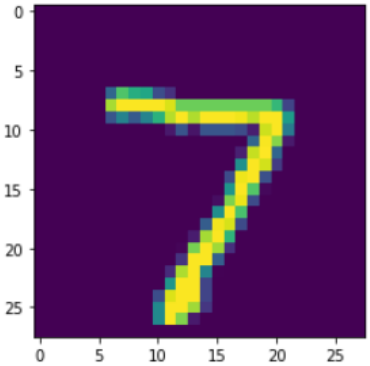

Figura 3.10: *Imagen que clasifica cómo un 7, y es un 7*

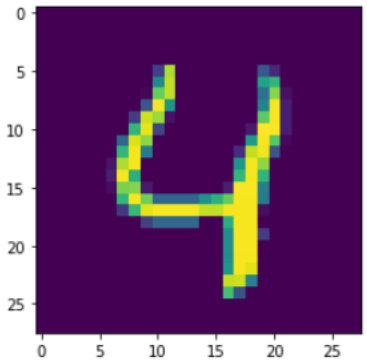

Figura 3.11: *Imagen que clasifica cómo un 4, y es un 4*

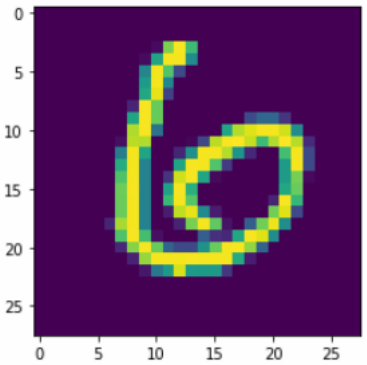

Figura 3.12: *Imagen que clasifica cómo un 6, y es un 6*

Bibliografía

[1] Libro: Random Number Generation and Montecarlo Methods página 6

[2] Libro: Random Number Generation and Montecarlo Methods página página 8

[3] Libro : Random Number Generation and Montecarlo Methods página página 25

[4] Números primos obtenidos de la página: https://asecuritysite.com/encryption/blum

[5] Libro: Random Number Generation and Montecarlo Methods página 42

[6] Libro: Random Number Generation and Montecarlo Methods página 48

[7] Todas las fórmulas de esta sección han sido extraidas del artículo: Deep Learning: An Introduction for Applied Mathematicians

[8] La demostración se encuentra en la página: https://towardsdatascience.com/derivative-of-the-sigmoid-function-536880cf918e

[9] Las demostraciones de las cuatro ecuaciones se pueden encontrar en el artículo Deep Learning: An Introduction for Applied Mathematicians, páginas 11,12,13

[10] Database: http://yann.lecun.com/exdb/mnist/