

Curso de Redes Neuronales y Backpropagation:

Cómo funciona el aprendizaje supervisado en una Red Neuronal

Datos importantes:

- Fue gracias a un par de científicos que hicieron experimentos sorprendentes con el cerebro de un gato, que se da pie a lo que llamamos **procesamiento de información visual**.
- Cuando una **neurona** transmite información a otra **neurona** en una **red biológica**, existe un proceso que es básicamente una *transferencia de electroquímicos* entre dos **neuronas**. Durante el proceso de esta transmisión existe una intensidad muy alta, sin embargo, antes y después de este proceso la intensidad es muy suave casi nula.
- El *cerebro humano* tiene distintos modos de procesar información. La profundidad en la cual queda procesada la información, es importante (**Deep Learning**)
- Una **Red Neuronal** es una composición de elementos *lineales* y *no lineales*, de tal manera que cuando se conectan y se entrena adecuadamente, permiten aproximar cualquier función matemática, es por esto que se le llama **aproximador de función universal**.
- El **aproximador de función universal** tiene la capacidad de separar clases de una forma *no lineal*.
- Las **neuronas** son unidades de información. Algo especial de las **neuronas** es que son básicamente *matrices*.
- Una **neurona** es una combinación lineal de entradas.

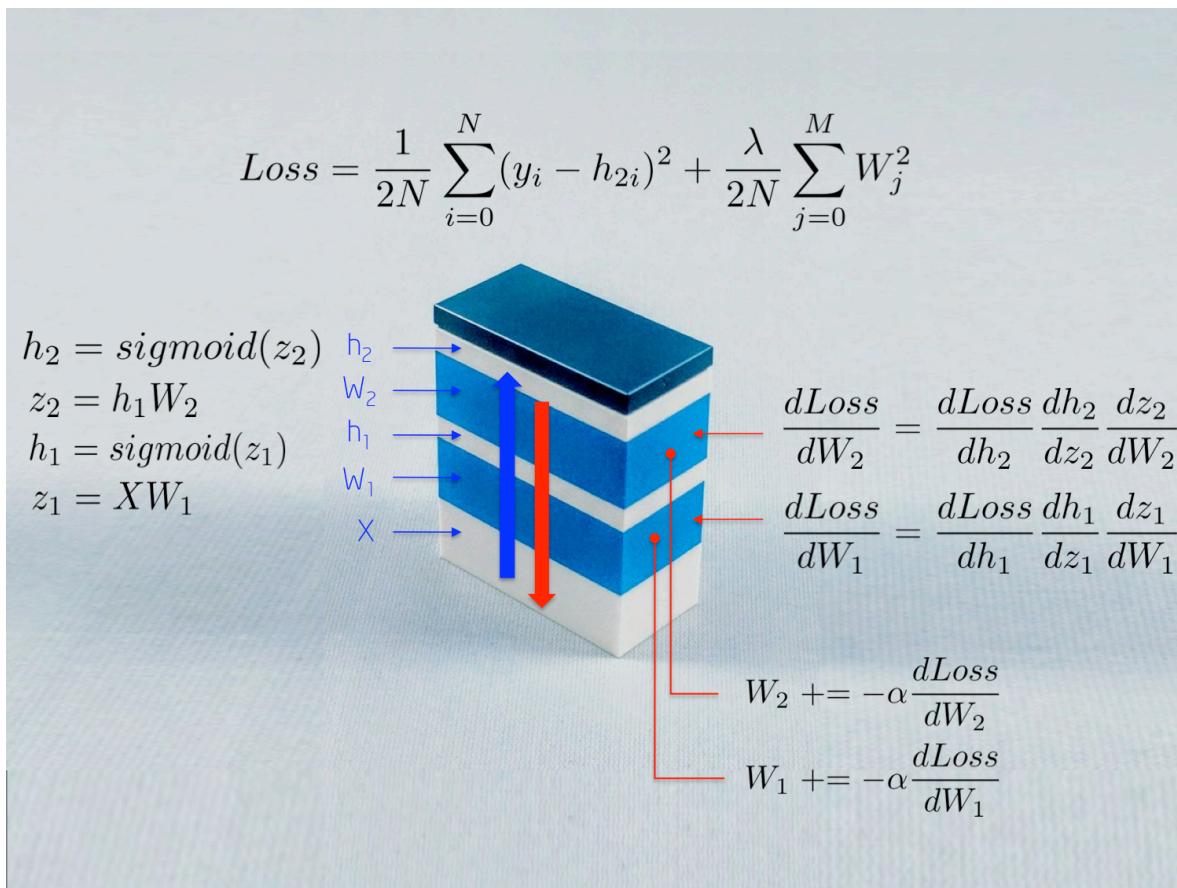
Explicando las matemáticas de cómo aprenden las Redes Neuronales

Una red neuronal es un composición inteligente de módulos lineales y no lineales. Cuando los escogemos sabiamente, tenemos una herramienta muy poderosa para optimizar cualquier función matemática. Por ejemplo una que separe clases con un límite de decisión no lineal.

Un tópico que no es siempre explicado en detalle, a pesar de su naturaleza intuitiva y modular, es el algoritmo de retro-alimentación (backpropagation algorithm), responsable de actualizar parámetros "entrenables" en la red. **Construyamos una red neuronal desde cero para ver el funcionamiento interno de una red neuronal** usando piezas de LEGO como una analogía, un bloque a la vez.

Puedes ver el código implementando estos conceptos en el siguiente repositorio: https://github.com/omar-florez/scratch_mlp

Las Redes Neuronales como una Composición de Piezas



La figura de arriba muestra algo de la matemática usada para entrenar una red neuronal. Haremos sentido de esto durante el artículo. Una red neuronal es una pila de módulos con diferentes propósitos:

- **Entrada X** alimenta la red neuronal con datos sin procesar, la cual se almacena en una matriz en la cual las observaciones con filas y las dimensiones son columnas.
- **Pesos W1** proyectan entrada X a la primera capa escondida h1. Pesos W1 trabajan entonces como un kernel lineal.
- **Una función Sigmoid** que previene los números de la capa escondida de salir del rango 0-1. El resultado es un array activaciones neuronales h1 = Sigmoid(WX).

Hasta este punto estas operaciones solo calculan un **sistema general lineal**, el cual no tiene la capacidad de modelar interacciones no lineales. Esto cambia cuando ponemos otro elemento en el pila, añadiendo profundidad a la estructura modular. Mientras más profunda sea la red, más interacciones no-lineales podremos aprender y problemas mas complejos podremos resolver, lo cual puede explicar en parte la popularidad de redes neuronales.

¿Por qué debería leer esto?

Si uno entiende las partes internas de una red neuronal, es mas fácil saber **qué cambiar primero** cuando el algoritmo no funcione como es esperado, y permite definir una estrategia para **probar invariantes y comportamientos esperados** que uno saben son parte del algoritmo. Esto también es útil **cuando quieres crear nuevos algoritmos que actualmente no están**

implementados en la **librería de Machine Learning de preferencia**.

¿Por qué hacer debugging de modelos de aprendizaje de máquina es una tarea compleja?

Por experiencia, los modelos matemáticos no funcionan como es esperado al primer intento. A veces estos pueden darte una exactitud baja para datos nuevos, tomar mucho tiempo de entrenamiento o mucha memoria RAM, devolver una gran cantidad de falsos negativos o valores NaN (Not a Number), etc. Déjame mostrarte algunos casos donde saber cómo funciona el algoritmo puede ser útil:

- Si toma mucho tiempo para entrenar, es quizás una buena idea incrementar el tamaño del mini-batch o array de observaciones que alimentan a la red neuronal, para reducir la varianza en las observaciones y así ayudar al algoritmo a converger.
- Si se observa **valores NaN**, el algoritmo ha recibido gradientes con valores muy altos produciendo desborde de memoria RAM. Piensa esto como una secuencia de multiplicaciones de matrices que explotan después de varias iteraciones. Reducir la velocidad de aprendizaje tendrá el efecto de escalar estos valores. Reduciendo el número de capas reducirá el número de multiplicaciones. Y poniendo una cota superior a los gradientes (clipping gradients) controlará este problema explícitamente.

Función de costo, supuestos y probabilidad

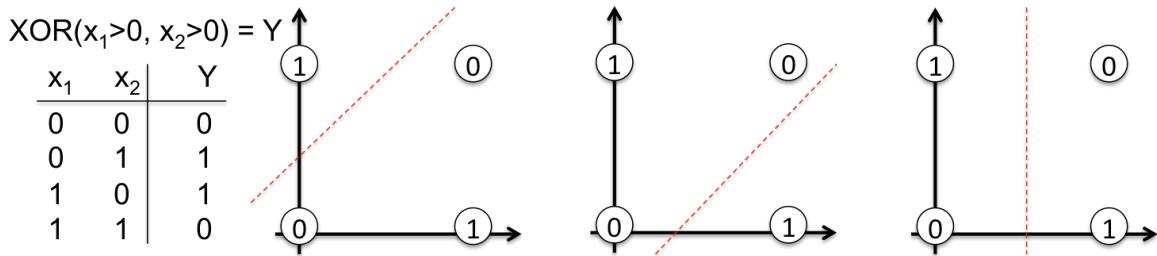
Datos importantes:

- Una función de error o *loss function* es básicamente la forma como una red neuronal sabe si está haciendo un buen trabajo o no.
- La gradiente o la primer derivada de una función con respecto a los pesos está siempre orientada al lugar donde se reduce más la función de error. Esto nos permite saber si la red neuronal está aprendiendo o no.
- En la vida real las funciones son no convexas, esto quiere decir que podemos tener distintas soluciones para un mismo problema. Este es uno de los motivos por los cuales toma tanto tiempo entrenar una red neuronal artificial.

Un Ejemplo Concreto: Aprendiendo la Función XOR

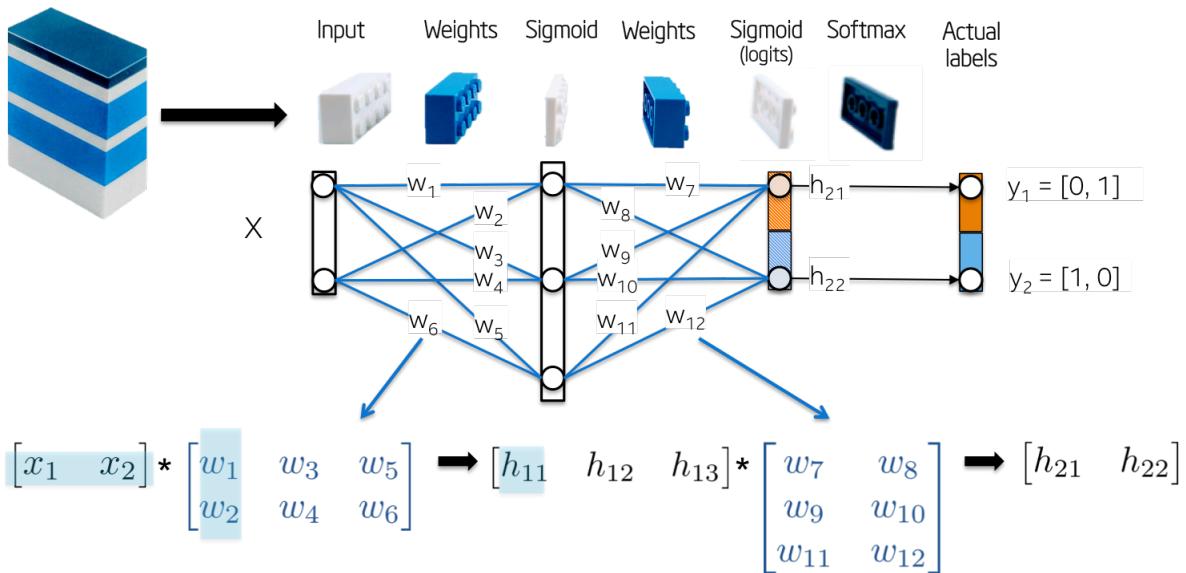
Abramos la caja negra. Construiremos a continuación una red neuronal desde cero que aprende la **función XOR**. La elección de esta **función no lineal** no es por casualidad. Sin backpropagation sería difícil aprender a separar clases con una **Línea recta**.

Para ilustrar este importante concepto, note a continuación cómo una línea recta no puede separar 0s y 1s, las salidas de la función XOR. **Los problemas reales también son linealmente no separables**.



La topología de la red es simple:

- Entrada X es un vector de dos dimensiones
- Pesos W1 son una matriz de 2x3 dimensiones con valores inicializados de forma aleatoria
- Capa escondida h1 consiste de 3 neuronas. Cada neurona recibe como entrada la suma de sus observaciones escaladas por sus pesos, este es el producto punto resaltado en verde en la figura de abajo: $z1 = [x1, x2][w1, w2]$
- Pesos W2 son una matriz de 3x2 con valores inicializados de forma aleatoria
- Capa de salida h2 consiste de 2 neuronas ya que la función XOR retorna 0 ($y1=[0,1]$) o 1 ($y2 = [1,0]$)
- Mas visualmente:

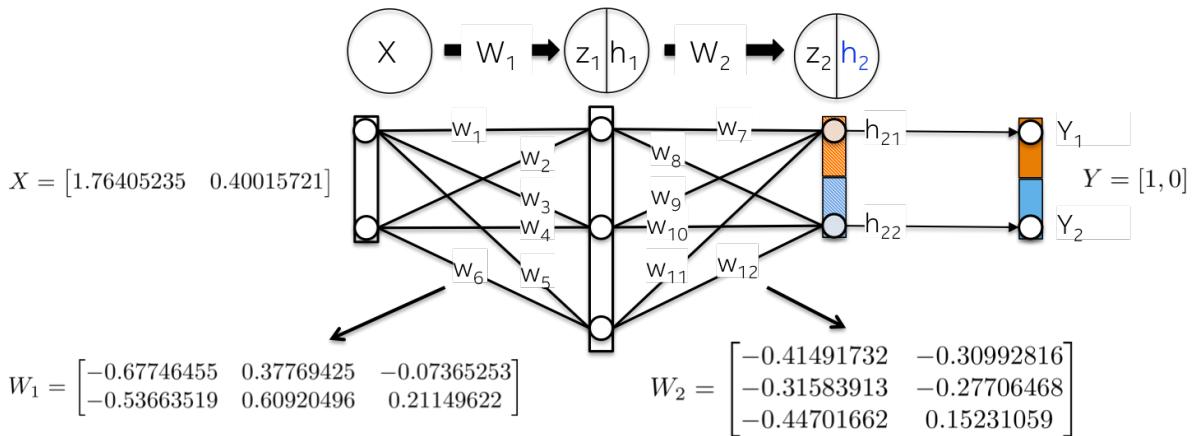


Entrenemos ahora el modelo. En nuestro ejemplo los valores entrenables son los pesos, pero tenga en cuenta que la investigación actual está explorando nuevos tipos de parámetros a ser optimizados. Por ejemplo: atajos entre capas, distribuciones estables en las capas, topologías, velocidades de aprendizaje, etc.

Backpropagation es un método para actualizar los pesos en la dirección (**gradiente**) que minimiza una métrica de error predefinida, conocida como **Función de Pérdida** o función de costo, dado un conjunto de observaciones etiquetadas. Este algoritmo ha sido repetidamente redescubierto y es un caso especial de una técnica más general llamada diferenciación automática en modo acumulativo reverso.

Inicialización de la red

Inicialicemos **los pesos de la red** con valores aleatorios.



Propagación hacia adelante:

El objetivo de este paso es propagar hacia delante la entrada X a cada capa de la red hasta calcular un vector en la capa de salida h2.

Es así como sucede:

- Se proyecta linealmente la entrada X usando pesos W1 a manera de kernel:

$$z_1 = XW_1$$

$$z_1 = [1.76405235 \quad 0.40015721] \begin{bmatrix} -0.67746455 & 0.37769425 & -0.07365253 \\ -0.53663519 & 0.60920496 & 0.21149622 \end{bmatrix}$$

$$z_1 = [-1.40982136 \quad 0.91005018 \quad -0.04529517]$$

- Se escala esta suma z1 con una función Sigmoid para obtener valores de la primera capa escondida. Note que el vector original de 2D ha sido proyectado ahora a 3D.

$$h_1 = \text{sigmoid}(z_1)$$

$$h_1 = [0.19626223 \quad 0.71301043 \quad 0.48867814]$$

- Un proceso similar toma lugar para la segunda capa h2. Calculemos primero la suma z2 de la primera capa escondida, la cual es ahora un vector de entrada.

$$z_2 = h_1 W_2$$

$$z_2 = [0.19626223 \quad 0.71301043 \quad 0.48867814] \begin{bmatrix} -0.41491732 & -0.30992816 \\ -0.31583913 & -0.27706468 \\ -0.44701662 & 0.15231059 \end{bmatrix}$$

$$z_2 = [-0.52507645 \quad -0.18394635]$$

- Y luego calculemos su activación Sigmoid. Este vector [0.37166596 0.45414264] representa el logaritmo de la probabilidad o vector predecido, calculado por la red dado los datos de entrada X.

$$h_2 = \text{sigmoid}(z_2)$$

$$h_2 = [0.37166596 \quad 0.45414264]$$

Calculando el error total

También conocido como “valor real menos predecido”, el objetivo de la función de pérdida es cuantificar la distancia entre el vector predecido h_2 y la etiqueta real proveída por un ser humano, y .

Note que la función de pérdida contiene un **componente de regularización** que penaliza valores de los pesos muy altos a manera de una regresión L2. En otras palabras, grandes valores cuadrados de los pesos incrementaran la función de pérdida, **una métrica de error que en realidad queremos reducir**.

Cuál es el algoritmo de retropropagación

Datos importantes:

- Los **gradientes** son básicamente la primer derivada de una función con respecto a un determinado parámetro. En este caso la función es la función de costo y el parámetro es cada uno de los pesos en una red neuronal.
- La técnica de **Backpropagation** es responsable de separar de forma no lineal distintas clases que se encuentran en la vida real.
- El **algoritmo de Backpropagation** se compone de dos pasos:
 - a) **Forward Step o paso hacia delante.** En éste tenemos una proyección de matrices. Tenemos datos de entrada, los pesos y una proyección de los datos de entrada a un nuevo espacio.
 - b) **Backward Step o paso hacia atrás.**
- La función de activación no lineal permite encontrar el valor de la primera capa escondida de una red neuronal y se hace una proyección con los pesos, para así obtener el valor sigmoid de esa combinación lineal.

Actualizar los pesos de la red neuronal utilizando gradientes

Datos importantes:

- El primer paso del **algoritmo de Backpropagation (paso hacia delante)** está orientado a poder permitir transformar los datos de entrada en la siguiente capa, que es la capa escondida, y finalmente transformar eso a la última capa que es la capa de salida.
- En el segundo paso del **algoritmo de Backpropagation (paso hacia atrás)** hacemos el proceso anterior pero a la inversa, comenzando por la capa de salida hasta llegar a la capa de entrada.

Propagación hacia atrás

El objetivo de este paso es **actualizar los pesos de la red neuronal** en una dirección que minimiza la función de pérdida. Como veremos mas adelante, este es un **algoritmo recursivo**, el cual reutiliza gradientes previamente calculadas y se basada plenamente en **funciones diferenciables**. Ya que estas actualizaciones reducen la función de pérdida, una red 'aprende' a aproximar las etiquetas de nuevas observaciones. Una propiedad llamada **generalización**.

Este paso va en **orden reverso** que la propagación hacia adelante. Este calcula la primera derivada de la función de pérdida con respecto a los pesos de la red neuronal de la capa de salida ($d\text{Loss}/dW_2$) y luego los de la capa escondida ($d\text{Loss}/dW_1$). Expliquemos en detalle cada uno.

dLoss/dW₂:

La regla de la cadena dice que podemos descomponer el calculo de gradientes de una red neuronal en **funciones diferenciables**:

$$\frac{d\text{Loss}}{dW_2} = \frac{d\text{Loss}}{dh_2} \frac{dh_2}{dz_2} \frac{dz_2}{dW_2}$$

$$\frac{d\text{Loss}}{dh_2} = -(y - h_2)$$

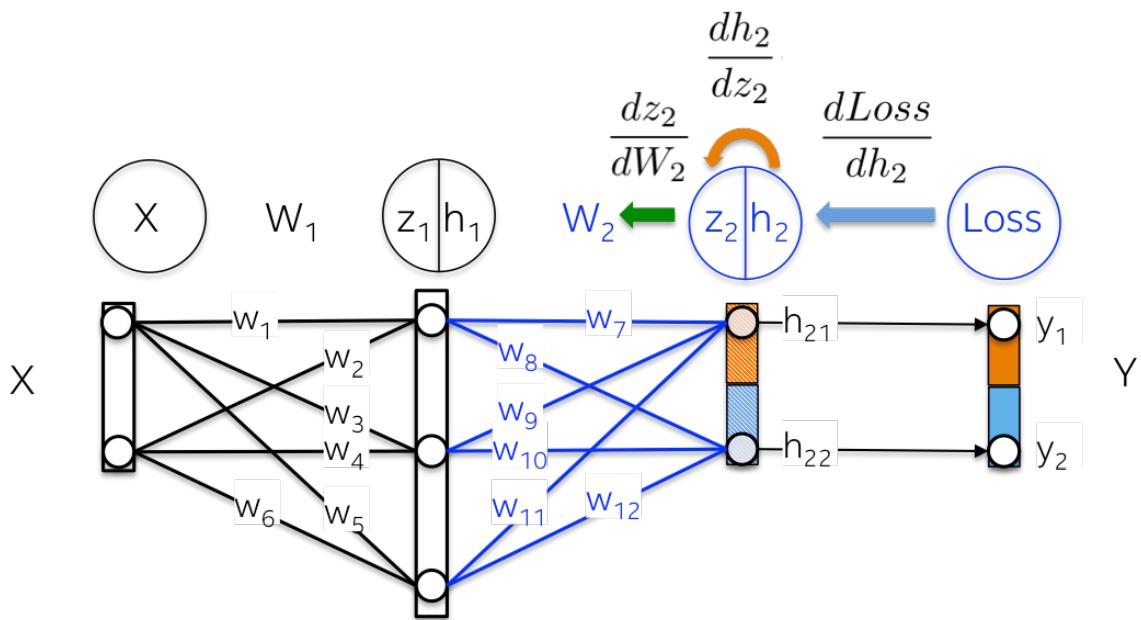
$$\frac{dh_2}{dz_2} = h_2(1 - h_2)$$

$$\frac{dz_2}{dW_2} = h_1$$

Aquí están las **definiciones de funciones** usadas arriba y sus primeras derivadas:

Función	Primera derivada
$\text{Loss} = (y-h_2)^2$	$d\text{Loss}/dW_2 = -(y-h_2)$
$h_2 = \text{Sigmoid}(z_2)$	$dh_2/dz_2 = h_2(1-h_2)$
$z_2 = h_1 W_2$	$dz_2/dW_2 = h_1$
$z_2 = h_1 W_2$	$dz_2/dh_1 = W_2$

Mas visualmente, queremos actualizar los pesos W_2 (en azul) en la figura de abajo. Para eso necesitamos calcular tres **derivadas parciales a lo largo de la cadena**.



Insertando esos valores esas derivadas parciales nos permiten calcular gradientes con respecto a los pesos W_2 como sigue.

$$\frac{dLoss}{dW_2} = \frac{dLoss}{dh_2} \frac{dh_2}{dz_2} \frac{dz_2}{dW_2}$$

$$\frac{dLoss}{dh_2} = -(y - h_2)$$

$$\frac{dh_2}{dz_2} = h_2(1 - h_2)$$

$$\frac{dz_2}{dW_2} = h_1$$

$$\begin{aligned} \frac{dLoss}{dh_2} &= -(y - h_2) \\ &= -([1 \ 0] - [0.37166596 \ 0.45414264]) \\ &= [-0.62833404 \ 0.45414264] \\ \frac{dh_2}{dz_2} &= h_2(1 - h_2) \\ &= [0.37166596 \ 0.45414264] (1 - [0.37166596 \ 0.45414264]) \\ &= [-0.23353037 \ 0.2478971] \\ \frac{dz_2}{dW_2} &= h_1 \\ &= [0.19626223 \ 0.71301043 \ 0.48867814] \end{aligned}$$

El resultado es una matriz de 3×2 llamada $dLoss/dW_2$, la cual actualizara los valores originales de W_2 en una dirección que minimiza la función de pérdida.

$$\frac{dLoss}{dW_2} = \frac{dLoss}{dh_2} \frac{dh_2}{dz_2} \frac{dz_2}{dW_2}$$

$\frac{dLoss}{dh_2}$ = $-(y - h_2)$
 $\frac{dh_2}{dz_2}$ = $h_2(1 - h_2)$
 $\frac{dz_2}{dW_2}$ = h_1

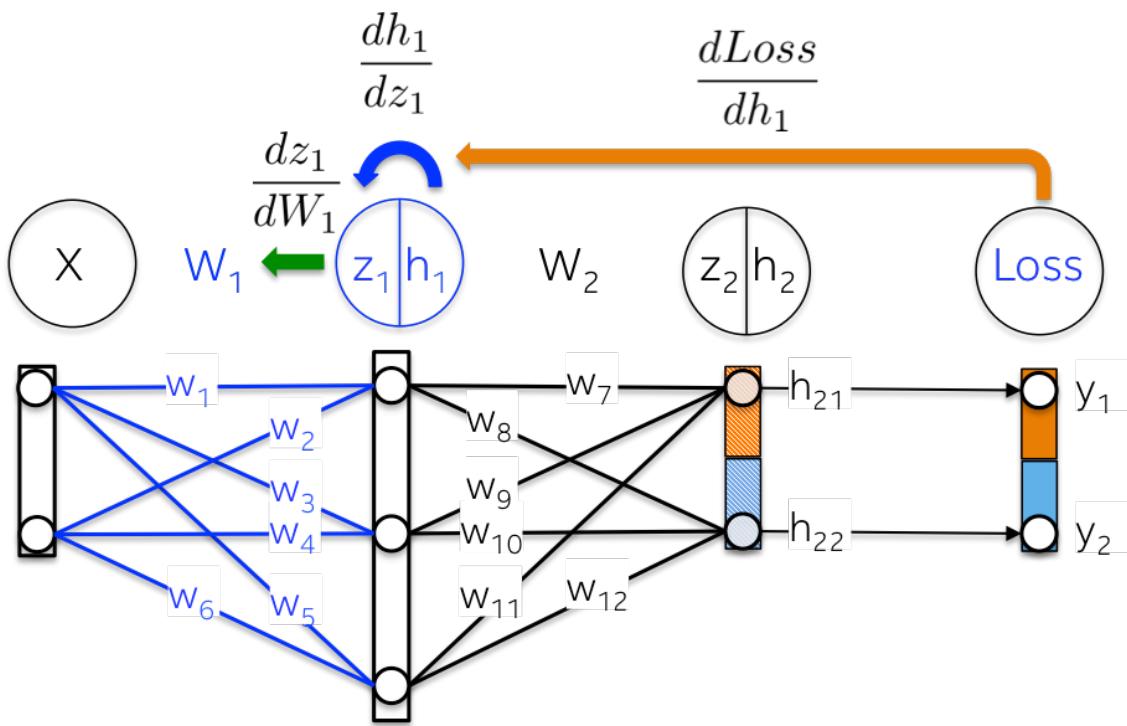
$$\begin{aligned} \frac{dLoss}{dW_2} &= [0.19626223 \quad 0.71301043 \quad 0.48867814]^T [-0.23353037 \quad 0.2478971] [-0.62833404 \quad 0.45414264] \\ &= \begin{bmatrix} -0.02879856 & 0.02209533 \\ -0.10462365 & 0.08027117 \\ -0.07170623 & 0.0550157 \end{bmatrix} \\ W_2^* &= W_2 - \alpha \frac{dLoss}{dW_2} = \begin{bmatrix} -0.41491732 & -0.30992816 \\ -0.31583913 & -0.27706468 \\ -0.44701662 & 0.15231059 \end{bmatrix} - 0.001 \begin{bmatrix} -0.02879856 & 0.02209533 \\ -0.10462365 & 0.08027117 \\ -0.07170623 & 0.0550157 \end{bmatrix} \\ W_2^* &= \begin{bmatrix} -0.41488852 & -0.30995026 \\ -0.31573451 & -0.27714495 \\ -0.44694491 & 0.15225557 \end{bmatrix} \end{aligned}$$

dLoss/dW1:

Calculando la **regla de la cadena** para actualizar los pesos de la primera capa escondida W1 exhibe la posibilidad de **reutilizar cálculos existentes**.

$$\begin{aligned} \frac{dLoss}{dW_1} &= \frac{dLoss}{dh_1} \frac{dh_1}{dz_1} \frac{dz_1}{dW_1} & \frac{dLoss}{dW_2} &= \frac{dLoss}{dh_2} \frac{dh_2}{dz_2} \frac{dz_2}{dW_2} \\ \frac{dLoss}{dh_1} &= \frac{dLoss}{dz_2} \frac{dz_2}{dh_1} \end{aligned}$$

Mas visualmente, el **caminio desde la capa de salida hasta los pesos W1** toca derivadas parciales ya calculadas en capas mas superiores.



Por ejemplo, la derivada parcial $d\text{Loss}/dh_2$ y dh_2/dz_2 ha sido ya calculada como una dependencia para aprender los pesos de la capa de salida $d\text{Loss}/dW_2$ en la sección anterior.

$$\begin{aligned}
 \frac{d\text{Loss}}{dW_1} &= \frac{d\text{Loss}}{dh_1} \frac{dh_1}{dz_1} \frac{dz_1}{dW_1} & \frac{d\text{Loss}}{dW_2} &= \frac{d\text{Loss}}{dh_2} \frac{dh_2}{dz_2} \frac{dz_2}{dW_2} \\
 \frac{d\text{Loss}}{dh_1} &= \frac{d\text{Loss}}{dz_2} \frac{dz_2}{dh_1} \\
 \frac{d\text{Loss}}{dh_1} &= \frac{d\text{Loss}}{dz_2} \frac{dz_2}{dh_1} \\
 &= [-(y - h_2)h_2(1 - h_2)][W_2] \\
 &= \begin{bmatrix} 0.23353037 & 0.2478971 \end{bmatrix} \begin{bmatrix} -0.41491732 & -0.30992816 \\ -0.31583913 & -0.27706468 \\ -0.44701662 & 0.15231059 \end{bmatrix} \\
 &= \begin{bmatrix} 0.02599101 & 0.01515256 & 0.08274025 \end{bmatrix} \\
 \frac{dh_1}{dz_1} &= h_1(1 - h_1) \\
 &= [0.19626223 \quad 0.71301043 \quad 0.48867814] (1 - [0.19626223 \quad 0.71301043 \quad 0.48867814]) \\
 &= [0.15774337 \quad 0.20462656 \quad 0.24987182] \\
 \frac{dz_1}{dW_1} &= X = [1.76405235 \quad 0.40015721]
 \end{aligned}$$

Ubicando todas las derivadas juntas, podemos ejecutar la **regla de la cadena** de nuevo para actualizar los pesos de la capa escondida W_1 :

$$\begin{aligned}\frac{dLoss}{dW_1} &= [1.76405235 \quad 0.40015721]^T \begin{bmatrix} 0.02599101 & 0.01515256 & 0.08274025 \end{bmatrix} \begin{bmatrix} 0.15774337 & 0.20462656 & 0.24987182 \end{bmatrix} \\ &= \begin{bmatrix} 0.00723246 & 0.00546965 & 0.03647082 \\ 0.00164061 & 0.00124073 & 0.00827303 \end{bmatrix}\end{aligned}$$

$$W_1^* = W_1 - \alpha \frac{dLoss}{dW_1} = \begin{bmatrix} -0.67746455 & 0.37769425 & -0.07365253 \\ -0.53663519 & 0.60920496 & 0.21149622 \end{bmatrix} - 0.001 \begin{bmatrix} 0.00723246 & 0.00546965 & 0.03647082 \\ 0.00164061 & 0.00124073 & 0.00827303 \end{bmatrix}$$

$$W_1^* = \begin{bmatrix} -0.67747178 & 0.37768878 & -0.073689 \\ -0.53663683 & 0.60920372 & 0.21148795 \end{bmatrix}$$

Finalmente, asignamos los nuevos valores de los pesos y hemos completado una iteración del entrenamiento de la red neuronal!

$$W_1 = W_1^*$$

$$W_2 = W_2^*$$

Demo: aprendiendo a separar clases

Datos importantes:

- Una de las premisas de usar redes neuronales es que finalmente tenemos una herramienta que permite separar clases de forma no lineal.
- Una sola línea no es capaz de poder separar las clases.

¿Qué necesitamos para implementar una **red neuronal** desde cero?

- Generar la topología de la red neuronal
- Representar los pesos
- Representar las activaciones no lineales
- Implementar la función de **Backpropagation**

Implementación y ejemplos de redes neuronales funcionando

Traduzcamos las ecuaciones matemáticas en código solamente utilizando Numpy como nuestro **motor de álgebra lineal**. Las redes neuronales son entrenadas en un loop en el cual cada iteración presenta **datos de entrada ya calibrados** a la red.

En este pequeño ejemplo, consideraremos todo el dataset en cada iteración. Los cálculos del paso de **Propagación hacia adelante, función de costo o pérdida, y Propagación hacia atrás** conducen a obtener una buena

generalización ya que actualizaremos los **parámetros entrenables** (matrices W1 and W2 en el código) con sus correspondientes gradientes (matrices dL_dw1 and dL_dw2) en cada ciclo.

El código está almacenado en este repositorio: https://github.com/omar-florez/scratch_mlp

```
# Initialize weights:
np.random.seed(2017)
w1 = 2.0*np.random.random((input_dim, hidden_dim))-1.0      #w0=(2,hidden_dim)
w2 = 2.0*np.random.random((hidden_dim, output_dim))-1.0      #w1=(hidden_dim,2)

#Calibrating variances with 1/sqrt(fan_in)
w1 /= np.sqrt(input_dim)
w2 /= np.sqrt(hidden_dim)

for i in range(num_epochs):
    index = np.arange(X.shape[0])[:N]

    # Forward step:
    h1 = sigmoid(np.matmul(X[index], w1))                      #(N, 3)
    logits = sigmoid(np.matmul(h1, w2))                         #(N, 2)
    probas = np.exp(logits)/np.sum(np.exp(logits), axis=1, keepdims=True)
    h2 = logits

    # Definition of Loss function: mean squared error plus Ridge regularization
    L = np.square(y[index]-h2).sum()/(2*N) + reg_coeff*(np.square(w1).sum()+np.square(w2).sum())/(2*N)

    # Backward step: Error = W_l e_l+1 f^l
    # dl/dw2 = dl/dh2 * dh2/dz2 * dz2/dw2
    dl_dh2 = -(y[index] - h2)                                     #(N, 2)
    dh2_dz2 = sigmoid(h2, first_derivative=True)                #(N, 2)
    dz2_dw2 = h1                                                 #(N, hidden_dim)
    #Gradient for W2: ((hidden_dim,N)x(N,2)*(N,2))
    dl_dw2 = dz2_dz2.T.dot(dl_dh2*dh2_dz2) + reg_coeff*np.square(w2).sum()

    #dl/dw1 = dl/dh2 * dh1/dz1 * dz1/dw1
    # dl/dh1 = dl/dz2 * dz2/dh1
    # dl/dz2 = dl/dh2 * dh2/dz2
    dl_dz2 = dl_dh2 * dh2_dz2                                    #(N, 2)
    dz2_dh1 = w2                                                 #(N, 2)
    dl_dh1 = dl_dz2.dot(dz2_dh1.T)                               #(N,2)x(2, hidden_dim)=(N, hidden_dim)
    dh1_dz1 = sigmoid(h1, first_derivative=True)                #(N,2)
    dz1_dw1 = X[index]                                           #(N,2)
    #Gradient for W1: (2,N)x((N,hidden_dim)*(N,hidden_dim))
    dl_dw1 = dz1_dw1.T.dot(dl_dh1*dh1_dz1) + reg_coeff*np.square(w1).sum()

    #weight updates:
    w2 -= -learning_rate*dL_dw2
    w1 += -learning_rate*dL_dw1
```

Input calibration

Forward

Loss function

Backward step

Weight updates

¡Ejecutemos esto!

Mire abajo **algunas redes neuronales** entrenadas para aproximar la **función XOR** en múltiple iteraciones.

Izquierda: Exactitud. **Centro:** Borde de decisión aprendido. **Derecha:** Función Loss.

Primero veamos como una red neuronal con **3 neuronas** en la capa escondida tiene una pequeña capacidad. Este modelo aprende a separar dos clases con un **simple borde de decisión** que empieza una línea recta, pero luego muestra un comportamiento no lineal. La función de costo en la derecha suavemente se reduce mientras el proceso de aprendizaje ocurre.

Teniendo **50 neuronas** en la capa escondida notablemente incremental el poder del modelo para aprender **bordes de decisión mas complejos**. Esto podría no solo producir resultados mas exactos, pero también **explorar las gradientes**, un problema notable cuando se entrena redes neuronales. Esto sucede cuando gradientes muy grandes multiplican pesos durante la propagación hacia atrás y así generan pesos actualizados muy grandes. Esta es la razón por la que **los valores de la función de costo repentinamente se incrementan** durante los últimos pasos del entrenamiento (step > 90). El **componente de regularización** de la función de costo calcula los **valores cuadrados** de los pesos que ya tienen valores muy altos ($\text{sum}(W^2)/2N$).

Este problema puede ser evitado **reduciendo la velocidad de aprendizaje** como puede ver abajo, implementando una política que reduzca la velocidad de aprendizaje con el tiempo, o imponiendo una regularización mas fuerte, quizás L1 en vez de L2. Los gradientes que explotan y se desvanecen son interesantes fenómenos y haremos un análisis detallada de eso en otra ocasión.