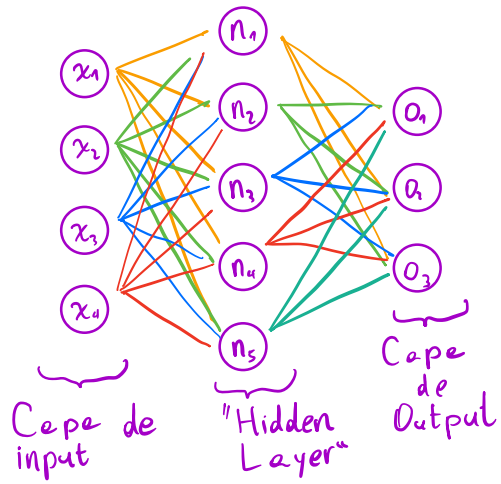
Redes Neuronales

Las Redes Neuronales son la herramienta más utilizada en el área de Machine Learning. La idea es (a grandes rasgos) replicar la forma de "aprender" de los humanos. Vamos a partir por un ejemplo. Supongamos el dataset Iris de 4 features $\{x_1, x_2, x_3, x_4\}$, en el que queremos clasificarlas en 3 tipos $\{1, 2, 3\}$. Una red neuronal se ve de la siguiente forma:

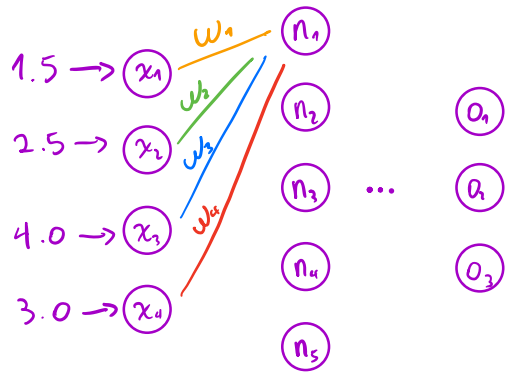


En donde tenemos una capa de **input**, una **"hidden layer"** y una capa de **output**. Hay que notar que la **"hidden layer"** se compone de las **neuronas** $\{n_1, \ldots, n_5\}$ y esta capa es una **"fully connected"** porque cada neurona se conecta a todos los nodos de la capa anterior y posterior

Ahora, ¿Cómo funciona esta red? Supongamos que

queremos clasificar una flor con los siguientes valores:

$$\{ x_1 : 1.5,\ x_2 : 2.5,\ x_3 : 4.0,\ x_4 : 3.0 \}$$



En la red, cada conexión entre capas tiene un peso. Aquí estamos mostrando los pesos de las conexiones entre la capa de input y la neurona n_1. Así, n_1 guardaría el valor:

$$V_{n_1} = w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4 x_4 + b_1$$

($x_1 \to 1.5$, $x_2 \to 2.5$, $x_3 \to 4.0$, $x_4 \to 3.0$)

Notamos que, además de los valores de las conexiones la neurona guarda un valor b_1, que llamamos "bias". Así, debido a la conexión de la capa de output y la "hidden layer" tenemos $4 \cdot 5 = 20$ pesos y 5 bias.

(4 inputs, 5 neuronas.)

Así, si llamamos w_{ij} el peso de la conexión entre la neurona i y el input j podemos definir la matriz M_1:

$$M_1 = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \\ w_{41} & w_{42} & w_{43} & w_{44} \\ w_{51} & w_{52} & w_{53} & w_{54} \end{bmatrix}$$

Y si consideramos el bias de cada neurona i como b_i y el output j como x_j, tenemos que:

$$\begin{bmatrix} V_{n_1} \\ V_{n_2} \\ V_{n_3} \\ V_{n_4} \\ V_{n_5} \end{bmatrix} = M_1 \vec{x} + \vec{b_1} \quad \text{con} \quad \vec{b_1} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \end{bmatrix} \quad \text{y} \quad \vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

De la misma forma, podemos definir M_2 como la matriz de 3 filas y 5 columnas con los pesos de las conexiones entre la "hidden layer" y la capa de output, así:

$$\begin{bmatrix} O_1 \\ O_2 \\ O_3 \end{bmatrix} = M_2 \cdot \begin{bmatrix} V_{n_1} \\ V_{n_2} \\ V_{n_3} \\ V_{n_4} \\ V_{n_5} \end{bmatrix} + \vec{b_2}$$

con $\vec{b_2}$ el vector de bias para estas conexiones.

Y la instancia nueva es clasificada con la clase representada por el o_k más grande.

Así, para este caso tenemos que aprender:

$4 \cdot 5 + 5 \cdot 3 = 20 + 15 = 35$ pesos

$5 + 3 = 8$ bias

Y una vez aprendidos estos valores, para clasificar solo debemos multiplicar matrices y sumar vectores.

Para entrenar una red neuronal se usa el algoritmo de Backpropagation.

Backpropagation

La técnica de Backpropagation nos sirve para ejecutar Gradient Descent de forma eficiente gracias a la regla de la cadena. No veremos los detalles técnicos, pero esta es la idea:

- Al inicio, inicializamos de manera aleatoria todos los pesos.
- Luego le mostramos un ejemplo etiquetado. Supongamos que es de clase 2 y el resultado fue el siguiente:

$O_1 = 0.5$

$O_2 = 0.35$

$O_3 = 0.15$

¡Pero nosotros queríamos este!

$O_1 = 0$

$O_2 = 1$

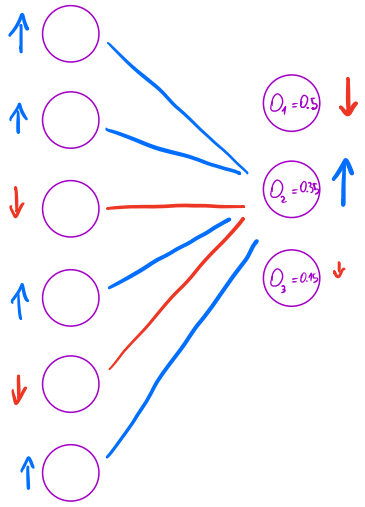
$O_3 = 0$

(Ojo, como en regresión softmax, en general optimizamos la función objetivo cross-entropy)

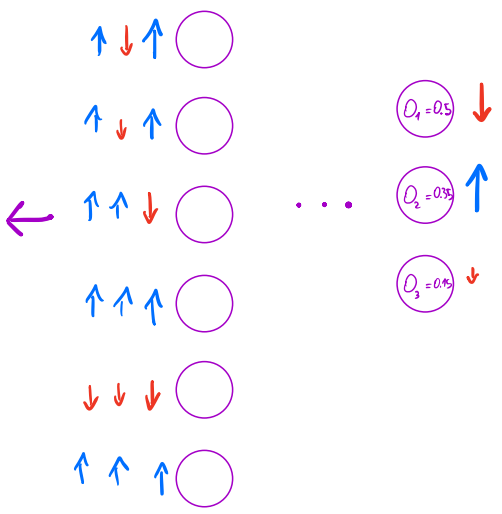
Así, le decimos a la capa de output que queremos que O_1 decrezca mucho, O_3 decrezca "poco" y O_2 incremente mucho:

Así, para que O_2 tenga un valor mayor, necesitamos:

- Incrementar el bias en O_2
- Incrementar los pesos de las conexiones a las neuronas con mayor valor numérico (i.e. neuronas "más activas")
- Pedirle a las neuronas conectadas a pesos negativos que se activen menos y a las con pesos positivos se activen más (proporcional al peso!)



Luego, juntamos los "deseos" de las otras neuronas de Output:



Y luego agregamos esos deseos y los pasamos a la hidden layer anterior

Después de esto, tendremos un valor numérico en el que queremos variar cada peso y bias de la red solo para este ejemplo. Luego, le deberíamos mostrar cada ejemplo para agregar los valores numéricos que cada ejemplo pide cambiar y así obtenemos el gradiente. En el entrenamiento, la red verá el dataset entero varias veces. Cada una de estas veces es una época (epoch). Los datos se reordenan aleatoriamente después de cada época. Y así, podemos entrenar nuestra red.