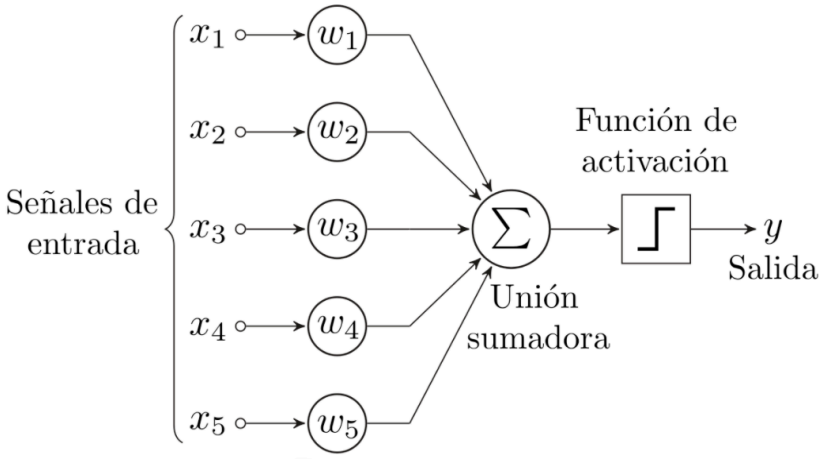
Redes neuronales

Ideas:

- Cada neurona es una unidad o componente, conectada con otra de distintas maneras
- Cada unidad recibe info de las neuronas que están conectadas a ellas (que vinieron antes)
- Cuando recibe esa info, hace un procesamiento interno con la info recibida y devuelve una salida que transmite a las neuronas con las que está conectada y las neuronas que vienen después.

Se puede modelar con un **perceptron**: Le asigna un peso a cada entrada dependiendo del resultado que se quiere obtener, las suma y le aplica una funcion de activacion y esa es la salida.



La función activación es lo que hace la neurona con las entradas que recibió y se puede decir que activa o no a la neurona.

Ejemplos de funciones de activación:

- No hacer nada, pasar directo (identity): **f(x) = x**
- Activacion linear (linear activation): **f(x) = a . x**
- T-step (con T y Threshold, valores elegidos a mano):
  f(x) = 0 si x <= threshold Threshold = valor umbral
  T si x > threshold T es un hiper parámetro. Si T es 1 se llama binary step

b → bias (que tan fácil es activar la neurona, cuanto mas grande mas facil de activarla).
b = - threshold.

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

Si hay muchos X nos queda como una regresion lineal o SVM. Hay un w y un b por cada X

$$\begin{bmatrix} w_{0,0} & w_{0,1} & \dots & w_{0,2} \\ w_{1,0} & w_{1,1} & \dots & w_{1,2} \\ \dots & \dots & \dots & \dots \\ w_{k,0} & w_{k,1} & \dots & w_{k,2} \end{bmatrix} \begin{bmatrix} a_0^{(0)} \\ a_1^{(0)} \\ \dots \\ a_n^{(0)} \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ \dots \\ b_n \end{bmatrix} = \begin{bmatrix} ? \\ ? \\ \dots \\ ? \end{bmatrix}$$

Pesos ✖ X ✚ Bias 〓 Y

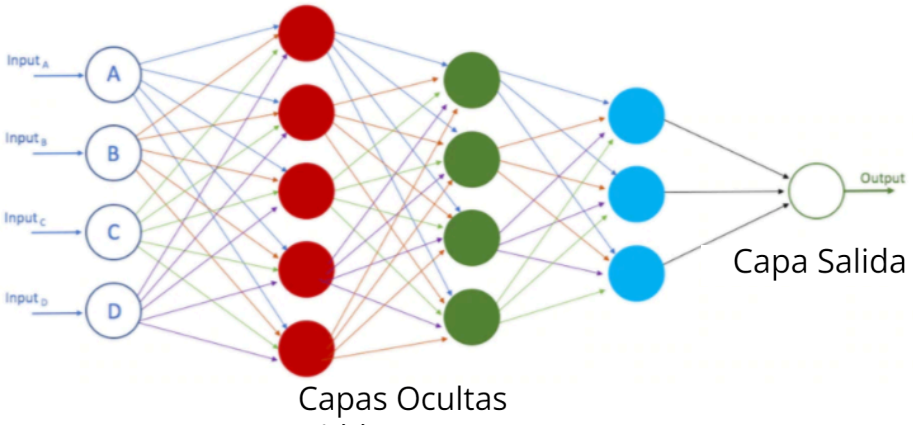
El perceptrón busca encontrar un hiperplano que divida las clases. Para los operadores lógicos, queremos que de un lado nos queden los unos y para el otro los ceros.

Perceptron no puede separar linealmente con la funcion de activacion T step casos como el XOR

Problema no linealmente separable

Puedo crear un sistema más complejo a partir de varios perceptrones.

Formas de unir perceptrones: le paso una función de activación y lo concateno a otro directamente pasandole la salida del perceptrón anterior (o multiplicando la salida por otro factor).



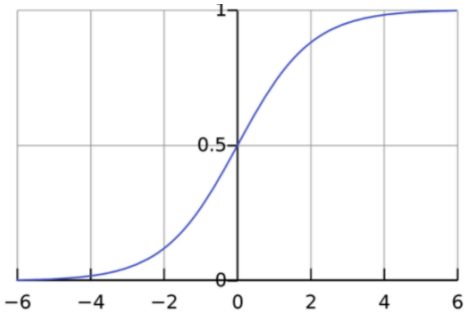
Voy a tener tantas neuronas en la última capa como clases haya.

Cada perceptron, dentro de las capas ocultas, tienen sus pesos bias y funcion de activacion.

Si unimos linealmente a los perceptrones, está demostrado que es lo mismo que tener uno solo.

Solución: en lugar de usar una función de activación lineal, podemos usar una que no lo sea. Función sigmoidea.

$$\sigma(z) \equiv \frac{1}{1+e^{-z}}.$$



La salida de un perceptrón entra como el z, y la salida es lo que recibe el siguiente perceptron.

CONVENCIÓN DE LA CÁTEDRA:

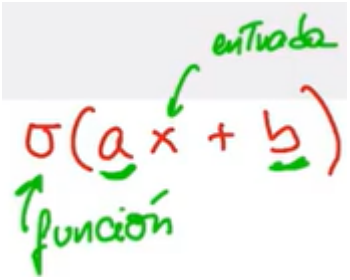
- Perceptron: t-step.
- Neurona: activación no lineal.
- Red neuronal: varias neuronas.

Luego de hacer la operación matricial, aplicamos la función sigmoidea.

$$\sigma\left(\begin{bmatrix} w_{0,0} & w_{0,1} & \dots & w_{0,2} \\ w_{1,0} & w_{1,1} & \dots & w_{1,2} \\ \dots & \dots & \dots & \dots \\ w_{k,0} & w_{k,1} & \dots & w_{k,2} \end{bmatrix} \begin{bmatrix} a_0^{(0)} \\ a_1^{(0)} \\ \dots \\ a_n^{(0)} \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ \dots \\ b_n \end{bmatrix}\right)$$

Entrenamiento de una red

Entrenar una red es encontrar el coeficiente a y el b, o los que haya, tales que mejore la predicción.



¿Cómo hacemos que la red aprenda? Parte de eso es comparar lo aprendido con lo real. **¿Cómo sabe si está aprendiendo bien o mal?** Necesitamos una forma de decirle dentro de la red, si la pifio o no.

Función de costo: Calcula la diferencia entre el valor predicho y el valor real y la neurona a la larga va a ajustar los parámetros para minimizar la función de costo.

¿Qué función de costo podemos usar?

Para las **regresiones** usamos la mean square error, **MSE**: $\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$

Para una **clasificación binaria** usamos la **binary-cross entropy**:

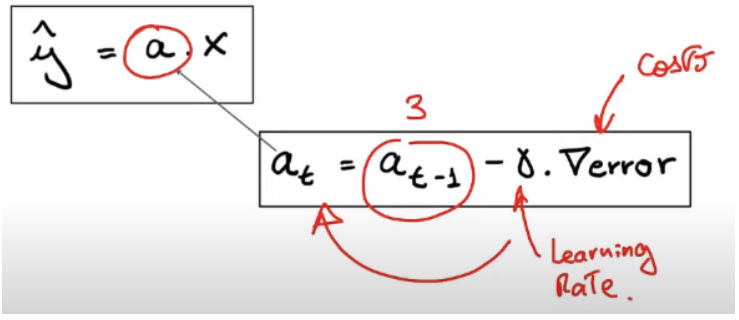
$$Loss = (Y)(-log(Y_{pred})) + (1-Y)(-log(1-Y_{pred})$$

Remains when Y = 1 | Removed when Y = 0 | Remains when Y = 0 | Removed when Y = 1

Para una **clasificación de M clases**, **cross entropy**: $-\frac{1}{m}\sum_{i=1}^{m} y_i \cdot \log(\hat{y}_i)$

¿Cómo actualizamos los pesos?

La red sabe que la pifio pero le falta el mecanismo para actualizar los pesos de acuerdo a cuando nos equivocamos. Esto se hace con **descenso por el gradiente**: le calculo la derivada a la función de costo y con eso actualizo w, ya que sabemos que en esa direccion se minimiza buscando converger en un mínimo local de la funcion de costo



El learning rate va a hacer que cambie mucho o poco en cada iteracion

¿Cada cuánto actualizo *a* con GD (gradient descent)?

- **GD**: espera a pasar por todos los puntos para poder actualizar la variable a. No se suele usar mucho porque es computacionalmente más costosa.
- **Mini batch GD**: Luego de cierta cantidad de valores actualiza. (ej: 36)
- **SGD**: Actualiza después de un valor.

Epochs: veces que se recorrió el data set.

Learning rate: cuanto me muevo a la dirección que dice el gradiente. Si es muy grande, vamos a oscilar. Si es muy chico tarda en converger.

¿Cómo hacemos para entrenar una red neuronal? Tenemos que entrenar todas las redes

Backpropagation: Una neurona depende de los valores de las activaciones de las neuronas de la capa anterior.

Buscamos modificar las activaciones anteriores, que fueron ocurriendo desde el final al inicio para minimizar el error. O sea, minimizar la función de costo. Todo para ajustar los pesos y los biases. A veces el error se maximiza en lugar de minimizarlo pero es por un menos.

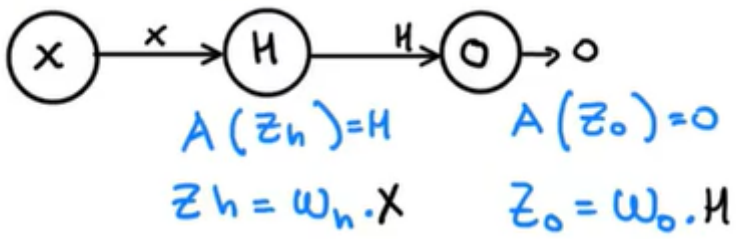
Seguimiento

H: neurona
H recibe solo a x. Lo multiplica por un coeficiente y le suma un bias (como toda neurona).
La salida de la neurona H es la entrada de otra neurona O. La salida de O es y techito.

Pasos:
1) Datos
2) Tipo de problema (para poder definir la arquitectura): puede ser de regresión o de clasificación.
3) Definir la entrada, la salida, las capas intermedias (hidden) -> arquitectura.
4) Métricas.

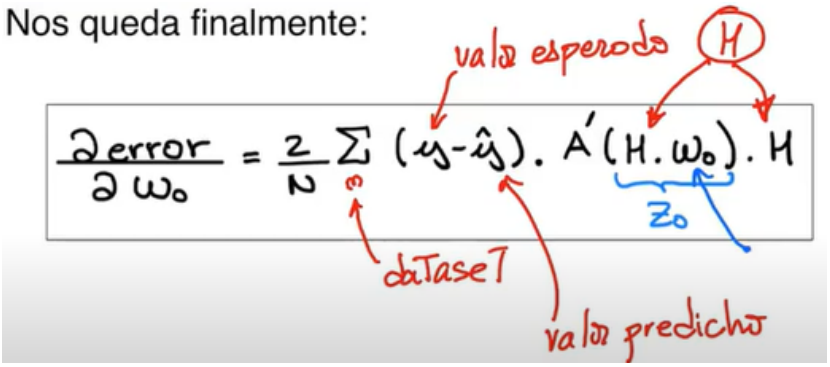
¿Que tengo que unir para que la arquitectura entrene con los datos? La función de costo o pérdida y optimizador para aplicar gradient descent.

Las métricas son parte de la verificación. Son parte de la evolución de cada epoch o iteración.



Lo que hacemos es ver que cambiar de esta arquitectura para minimizar la función de costo. Como la arquitectura ya está definida, solo puedo cambiar los coeficientes y biases que tiene. No puedo cambiar la función de activación.

Es lo mismo que la clase pasada solo que esta vez para una red.



Hay mucho laburo en cómo buscar los valores para inicializar los coeficientes.

¿Qué función de activación se usa en la última capa **cuando tengo una clasificación multiclase**? Se usa **softmax**.

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

Me da como output el scoring de que ese dato pertenezca a una clase.

Una característica de la softmax es que cuando tengamos que decir que clase es, devolvemos un vector que cada clase corresponde a una clase de scoring de que tantas chances hay de que sea de esa clase vs del resto.

Podemos ver que esta función lo que trata de hacer es darle mucha importancia a una de las clases y tratar de que el resto sea cero. De forma matemática está diciendo “elegí una”. Se elige la posición donde está el mayor valor.

La suma del vector da 1. Se puede relacionar con probabilidad en algunos casos.

Si la red es muy compleja: overfitting.

¿Cómo lo soluciono? **Regularización**. La derivada del error contempla ese coeficiente adicional. Cada error tiene la suma de la regularización.

Métodos de regularización:

Métodos que ayudan a la generalización. O sea, que funcione mejor con datos que nunca vio.

Regularización L1 y L2

La idea es penalizar el valor de los pesos en la red. L2 es por el módulo al cuadrado y L1 es por el módulo.

Hay que tener cuidado con el porcentaje de regularización, capaz el modelo le sale más barato no tener peso en ningún coeficiente y no soluciona el verdadero problema.

Dropout

La idea es ir apagando conexiones entre capas aleatoriamente durante el entrenamiento, de manera de evitar *codependencias* entre conexiones.

La idea es que aprenda algo útil durante las conexiones del momento.

Early stopping

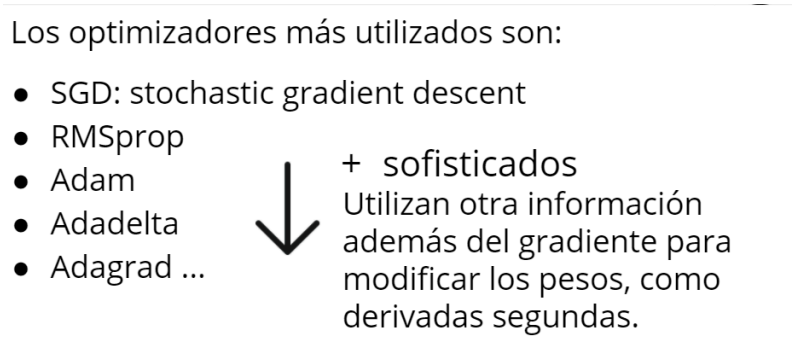
El método corta el entrenamiento cuando empieza a empeorar el error del set de validación, el error de generalización o el error con datos que nunca vio.

Data augmentation

La idea es agregar datos usando los datos que se tienen. La idea es modificar los datos que ya tenemos y sigan siendo verosímiles al problema que estoy resolviendo. Los puedo modificar aplicando transformaciones a los datos que ya tengo. Los datos siguen siendo útiles. Lo difícil es encontrar las transformaciones.

Puede ser para cualquier dato, no solo imágenes. Se aplican perturbaciones chicas. Si no terminamos entrenando el modelo con basura.

Optimizadores



Hiperparámetros

Son parámetros que son importantes para el modelo y en general son cosas que yo tengo que determinar antes de empezar a entrenar. Por ejemplo, el learning rate.

Entrenamiento de la red

Necesitamos:

- Arquitectura
- Hiperparametros
- Optimizador
- Funcion de perdida
- Funciones de activación