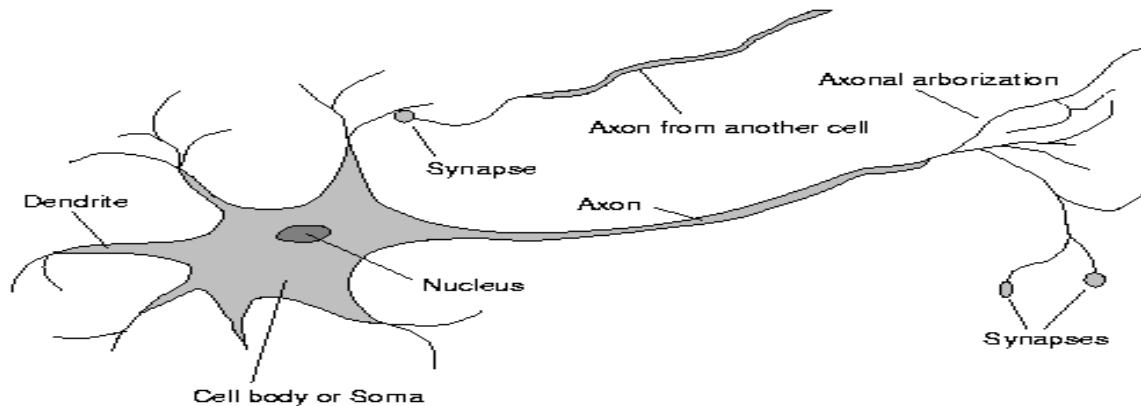


Introducción a las Redes Neuronales y Aprendizaje Profundo

Prof. Wílmer Pereira

Redes Neuronal Natural

Estructura celular del cerebro donde residen las capacidades intelectuales del hombre.
Desde 100×10^9 hasta 10×10^{12} neuronas ...



- Interneuronas
- Neuronas motoras (directo al músculo)
- Neuronas receptoras (directo desde el órgano sensor)

Neurona:

Célula nerviosa donde se procesan reacciones químicas

Núcleo:

Codifica (ADN) el funcionamiento de la neurona

Dendritas:

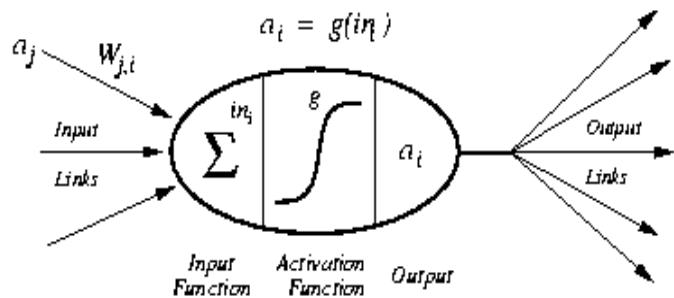
Ramificaciones entre neuronas que representan la entrada

Axón:

Prolongación de la neurona que transmite la activación o salida

Sinapsis:

Punto de unión entre dendritas donde ocurren las reacciones



- Las entradas (a_i) representan las dendritas
- La función de entrada (Σ) sintetiza las reacciones químicas con una suma ponderada
- La función de activación (g) es el disparador de la neurona
- La salida es el axon que transmite la respuesta

Propiedades de la Red Neuronal Natural



Plasticidad:

Nexos entre neuronas que se fortalecen (temporal o permanentemente) con los patrones de estímulo que generan proteínas en la neurona y la cambian. Esto representa los pesos en la neurona artificial (w_{ij})



Elasticidad:

Capacidad de crecer para agregar propiedades intelectuales. **No es dinámico en las redes neuronales artificiales**

La activación en una neurona artificial depende de una función que debe ser continua y derivable



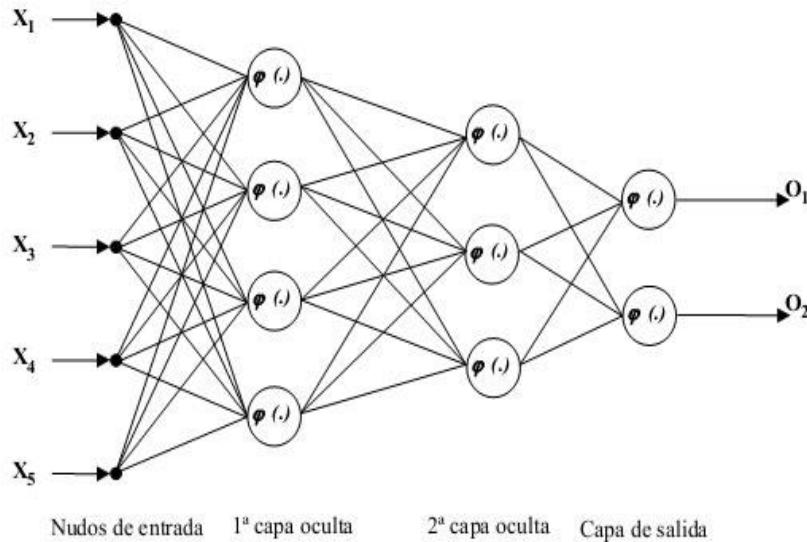
La especialización por capas de una red neuronal artificial en las primeras propuestas (redes neuronales superficiales) no era explícita. En las redes neuronales profundas hay una especialización estratificada ... No obstante, sigue siendo un modelo de caja negra pues ninguna red neuronal tiene capacidad de explicación como si la tienen los árboles de decisión.

Cerebro vs Computadora

- Almacenamiento:
Más neuronas que bits aunque la evolución computacional es vertiginosa, mucho mayor que la evolución de cerebro.
- Velocidad.
Computadora orden de los η seg
Cerebro del orden de los mseg
pero ... el cerebro es masivamente paralelo y en definitiva el cerebro es 10^{10} veces más rápido
- Tolerancia a fallas:
Una neurona natural dañada afecta de manera marginal el comportamiento del cerebro. En cambio, cualquier mínimo error altera todo el procesamiento a nivel de la computadora
- Complejidad de ejecución:
El cerebro realiza tareas muy complejas que son sencillas al humano pero difíciles para cualquier computadora
- Procesamiento:
Centralizado vs Distribuido
Computadora Cerebro
- Aprendizaje:
En el cerebro es *online* mientras que en la computadora frecuentemente es *offline*. Sin embargo, en ambos es distribuido

Redes Neuronales Artificiales

Unidades enlazadas a través de conexiones
cargadas por pesos numéricos w_{ij}

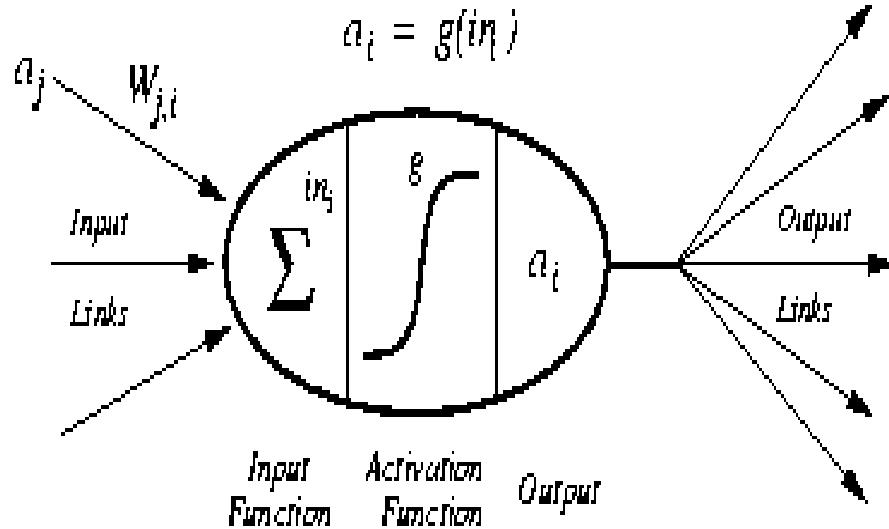


Frecuentemente la unión entre capas es densa, es decir, todas de las neuronas de una capa tienen conexión con todas las neuronas de la capa anterior

Una red puede tener una o varias salidas dependiendo de si es un clasificador binario o multiclase. También puede tener una salida continua (regresor).

- El nivel de activación de la neurona artificial (equivalente al impulso excitatorio) es un cálculo individual en cada neurona, sin control global (distribuido)
- El aprendizaje se basa en la actualización de esos pesos que se inicializan aleatoriamente pero se ajustan en la fase de entrenamiento de la red para acoplarse a los datos de entrada. Este proceso se llama la **retropropagación** (*backpropagation*) y se configura con múltiples hiperparámetros.

Funciones de Propagación



Suma Ponderada sin sesgo

$$in_i = \sum w_{ji}a_j$$

Distancia Euclídea

$$in_i = \sum (a_j - w_{ji})^2$$

Ponderada con sesgo

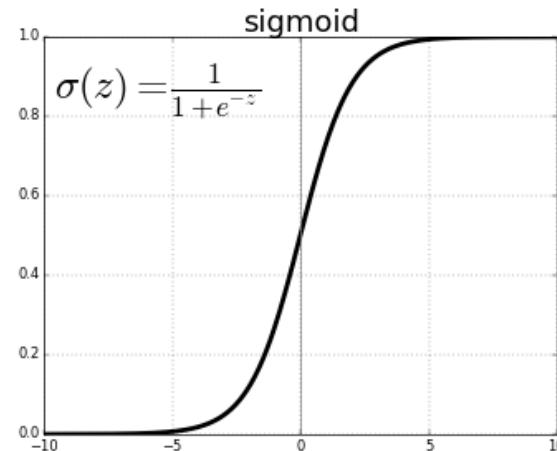
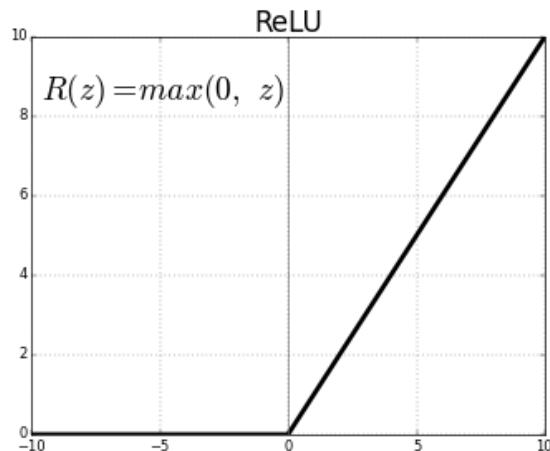
$$in_i = \sum w_{ji}a_j - b_i$$

Manhattan, Sigma-Pi, ...

- La función de entrada más común es la suma ponderada que realmente es la ecuación de un hiperplano donde b_j es el sesgo para la activación o inhibición de la neurona.
- La función de entrada, activación y salida deben estar separadas.
- La función salida normalmente es la identidad $f(x) = x$ aunque el disparo de la neurona puede depender de una probabilidad

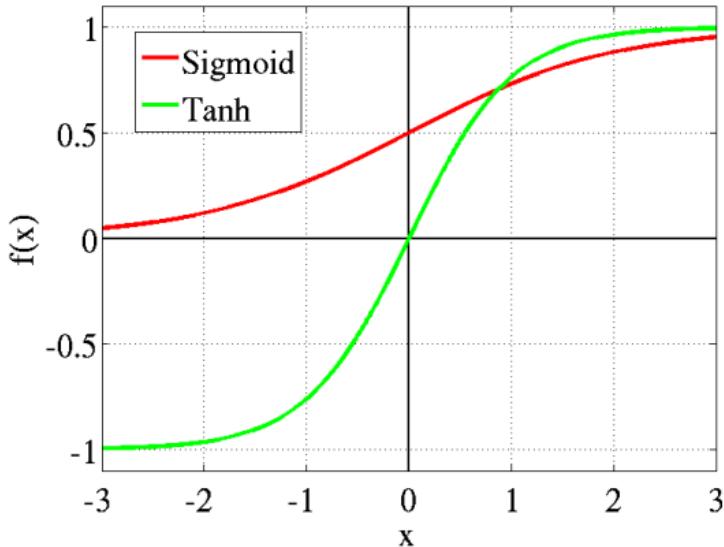
Funciones de Activación (g)

Función que deben tener todas las neuronas artificiales y que determina el valor de salida, por neurona, dado los estímulos de entrada



- La función de activación debe ser monótona, creciente, continua y derivable
- Además de sigmoide y ReLU (con sus variantes) están: softmax, tangente hiperbólica, maxcut, ...
- La función de activación puede ser diferente en cada neurona aunque lo común es que sea la misma función de activación por capa. Frecuentemente en las capas intermedias es ReLU y en la última capa se usa sigmoide (clasificación binaria), softmax (clasificación multiclase) y lineal (regresor).

sigmoide, tanh y softmax



En algunos casos particulares, para redes neuronales recurrente, Tanh es preferida sobre sigmoide. Sin embargo, en la mayoría de los casos, sigmoide se comporta mejor ...

La sigmoide, en general, se utiliza para modelar propagación de epidemias, difusión en redes sociales, ...

$$\sigma(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}}$$

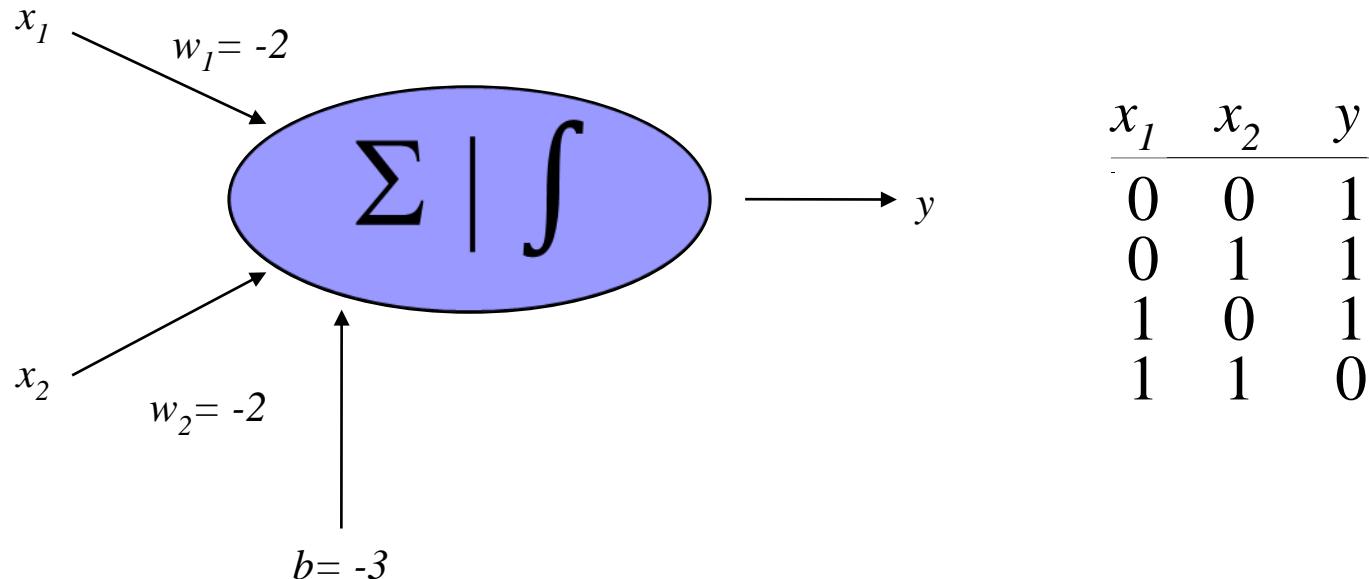
softmax aplica para clasificar cuando las categorías son mutuamente exclusivas por lo que es una generalización de la sigmoide.

No es fácil de graficar porque es una función multidimensional ...

Poder de una Red Neuronal

Como una red neuronal puede representar un NAND entonces, puede a su vez, representar cualquier función booleana, es decir, son capaces de modelar cualquier circuito digital combinatorio (sin retroalimentación)

Sea una red de una sola capa y una sola neurona con función de activación escalón



Sin embargo esto no significa que una red de una capa modele cualquier tipo de función. Por ejemplo, no necesariamente modela un circuito digital secuencial (con memoria)

Tipos de Aprendizaje



Supervisado:

La red requiere *feedback* de datos clasificados mientras se entrena mediante retropropagación, ajustando el error cometido en la extrapolación. Así se comportan la mayoría de las redes neuronales



No Supervisado o Autoorganizado:

La red reconoce la irregularidades del conjunto de entrada, extrae rasgos y los agrupa por similitud (*cluster*). Existen reded neuronales no supervisadas (mapas de Kohonen)



Híbrido:

Coexistencia de supervisado y autoorganizado. Es posible en las redes neuronales



Reforzado:

Se tiene información del error más no de la salida. Con la información del buen o mal comportamiento se ajusta la red. En principio, las redes neuronales no tienen este comportamiento (por reforzamiento)

Red Neuronal Supervisada

El objetivo es construir una función multivariable desconocida $f: \mathbb{R}^n \rightarrow \mathbb{R}^m$ a partir de la entrada $x \in \mathbb{R}^n$ y la respuesta $y \in \mathbb{R}^m$.

Así se minimiza iterativamente el error mediante aproximación estocástica. El error es una función de comparar y contra \hat{y} (predicción)

El proceso está conformado por cuatro fases:

Separación: Dividir los datos de entrada en conjunto de entrenamiento y conjunto de prueba. Es común usar también un conjunto de validación.

Entrenamiento: A partir de casos (x, y) conocidos modificar los pesos para minimizar el error entre y y \hat{y} (retropropagación)

Prueba: Durante el aprendizaje (validación) o a posteriori, verificar el ajuste de la red ante nuevos casos (x, y) no usados para el entrenamiento

Operación: Una vez que la red ha aprendido correctamente, según los resultados de la fase de prueba, colocar la red en ejecución.

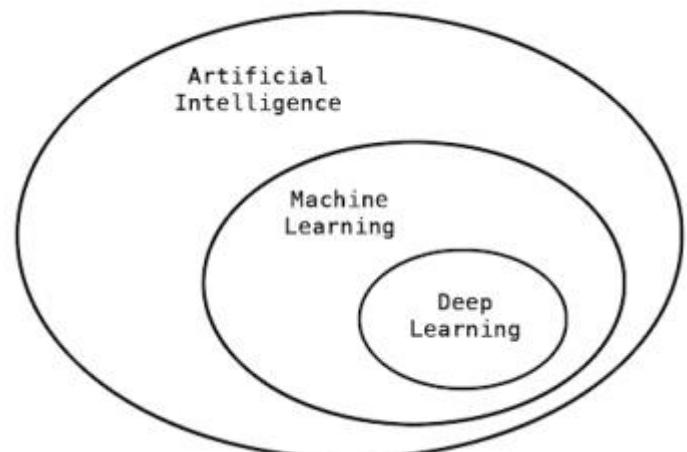
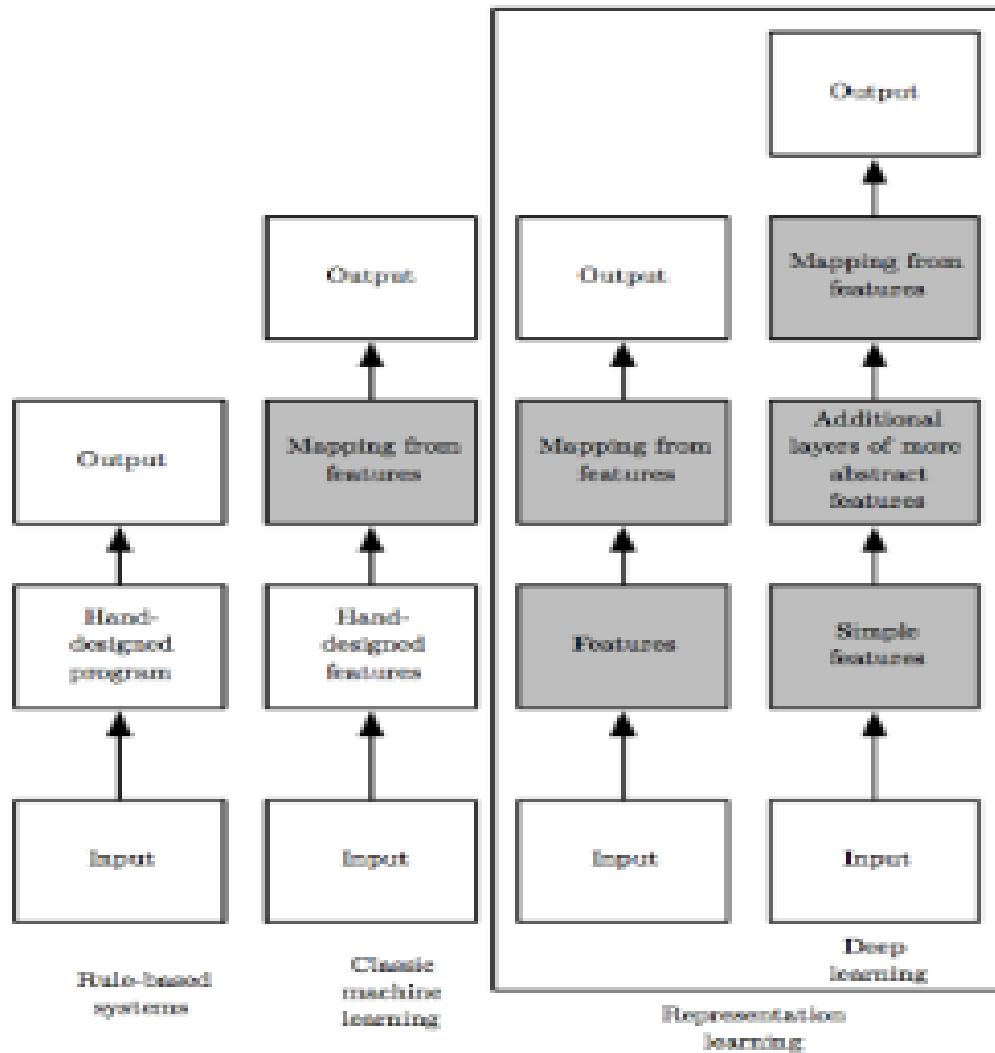
Todo este procesamiento es posible con `scikit learn` ([ejemplo](#)) entonces ¿Qué ofrece de nuevo *deep learning*?

Deep Learning

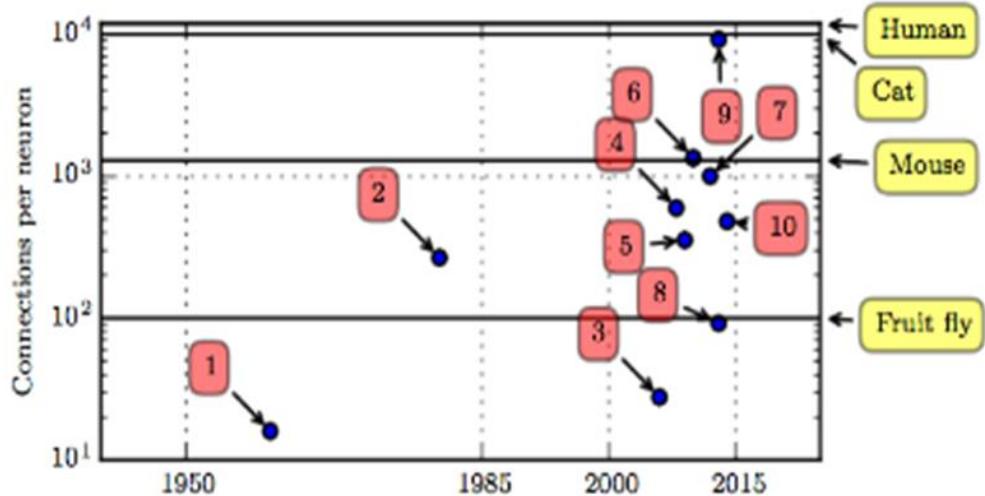
Esta propuesta se basa en una cantidad considerable de capas, inicialmente inspirado en las redes neuronales superficiales (pocas capas ocultas) pero ... progresivamente se ha alejado del modelo neuronal ...

- El énfasis está en colocar sucesivas capas de representación para procesar la información, por lo que el modelo de base es la red neuronal pero *deep learning* no siempre sigue el modelo del cerebro ...
- La gran cantidad de capas eran una limitación hace unos pocos años (*overfitting* y desvanecimiento de gradiente). De hecho alrededor del 2010 muy pocos investigadores trabajaban en redes neuronales
 - ... pero ...
 1. Un grupo de investigadores de IDSIA crearon una inmensa base de datos de imágenes ([ImageNet](#)) y la aparición de las redes neuronales convolucionales, dieron un vuelco al procesamiento digital de imágenes y las competencias de [Kaggle](#)
 2. El enorme crecimiento de número de procesadores que ofrecen las GPU y más recientemente las TPU, así como las plataformas en la nube de procesamiento paralelo, Azure, AWS y Google *cloud* (más específicamente [google colab](#)) ha favorecido el florecimiento del *deep learning*.
 3. Las redes neuronales superficiales no competían con *random forest* y SVM, pero las mejoras en la retropropagación, propiciaron los avances en *deep learning*...

Diferencias entre IA convencional, Machine Learning y Deep Learning



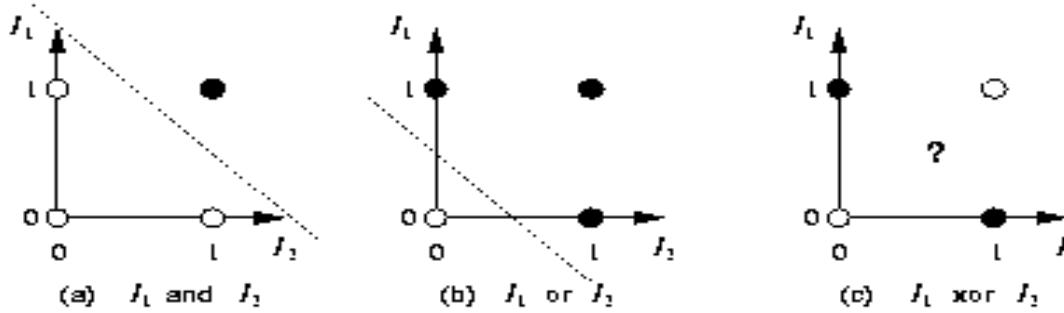
Evolución de la cantidad de neuronas en Deep Learning



1. Adaptive linear element (Widrow and Hoff, 1960)
2. Neocognitron (Fukushima, 1980)
3. GPU-accelerated convolutional network (Chellapilla et al., 2006)
4. Deep Boltzmann machine (Salakhutdinov and Hinton, 2009a)
5. Unsupervised convolutional network (Jarrett et al., 2009)
6. GPU-accelerated multilayer perceptron (Ciresan et al., 2010)
7. Distributed autoencoder (Le et al., 2012)
8. Multi-GPU convolutional network (Krizhevsky et al., 2012)
9. COTS HPC unsupervised convolutional network (Coates et al., 2013)
10. GoogLeNet (Szegedy et al., 2014a)

Aprendizaje en redes neuronales

- La primera propuesta de red neuronal (perceptrón: a una capa, sólo de salida) se demostró que sólo podía modelar problemas linealmente separables (Minski&Papert 1969). Aún con muchas capas, según ellos, no se solucionaba el problema



... sin embargo ...

- Alrededor de ese mismo año Byron&Ho publicaron el algoritmo para entrenar redes neuronales multicapas con retropropagación usando **descenso de gradiente** ... nadie les prestó atención y sólo 10 años después se redescubrió el algoritmo 😞 ...

... de nuevo ...

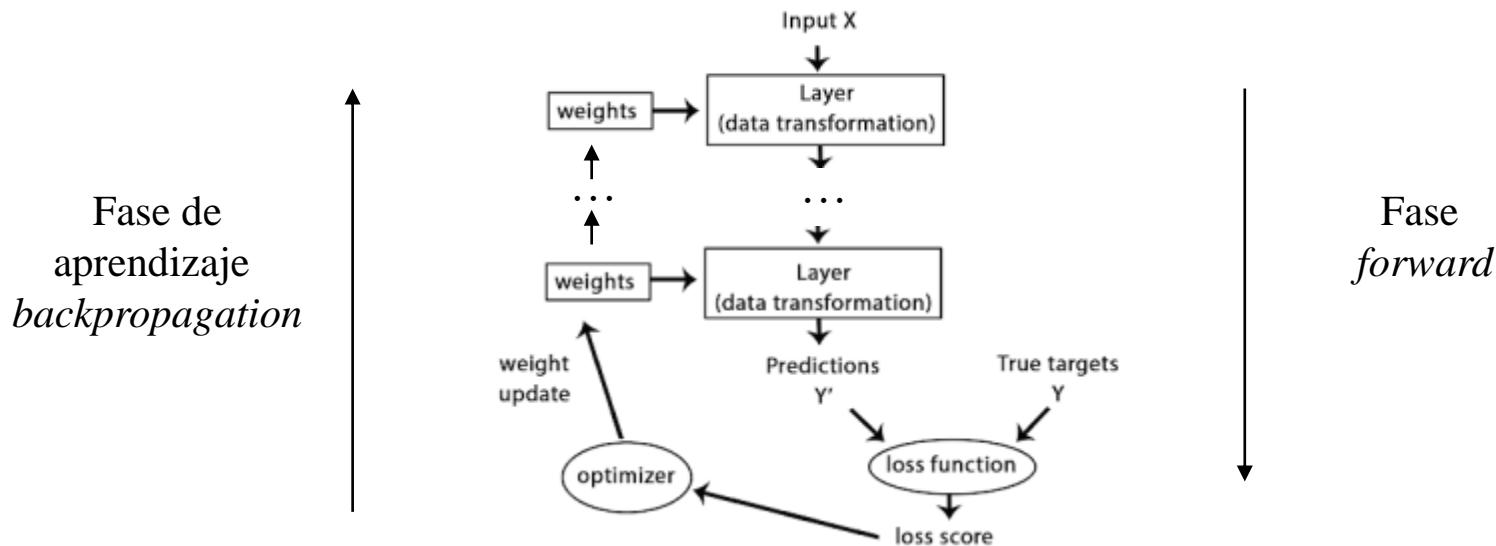
- Por el año 2010 el área estaba de nuevo estancándose, hasta que varios investigadores canadienses, franceses y norteamericanos, Hinton, Bengio, LeCun entre otros, reactivaron el área.
- El algoritmo de aprendizaje, clave para la eficiencia del área, es descenso de gradiente, con una gran cantidad de variantes ...

Librerías de deep learning

- Las librerías simples y con capacidad de paralelización automática, han impulsado la democratización del área. Además la comunidad *deep learning* publica sus resultados en [arXiv](#) antes de presentarlo en conferencias lo que facilita la obtención de información. También ofrecen [playground tensorflow](#).

Keras es una librería de alto nivel ([librería MIT](#)) para aprendizaje profundo que puede estar sobre Tensor Flow, Theano o CNTK. Se ejecuta sobre cluster usando [eigen](#) o sobre GPU mediante [cuDNN](#) (librería de CUDA aceleradora de GPU para *deep learning*)

- En *deep learning*, además de las diferentes funciones de activación, también hay varios tipos de funciones de pérdida y una gran variedad de optimizadores para parametrizar las implementaciones. En líneas generales el proceso se puede resumir como:



Ejemplo simple con Keras

- Una vez importada las clases necesarias y cargada la base de conocimiento para el entrenamiento de la red neuronal, se definen las capas y cantidad de neuronas por capas de la red

```
from keras import models
from keras import layers
...
network=models.Sequential()
network.add(layers.Dense(512,activation='relu',input_shape=(784,)))
network.add(layers.Dense(10,activation='softmax'))
```

- A continuación, se fija el algoritmo de backpropagatió (*optimizer*) a utilizar, la función de pérdida (*loss*) a minimizar con *backpropagation* y las métricas para medir el aprendizaje con el conjunto de prueba y el conjunto de validación.

```
network.compile(optimizer='sgd',
                 loss='categorical_crossentropy',
                 metrics=['accuracy']))
```

- Luego se entrena la red, determinando al cantidad de pasada de entrenamiento con el conjunto de entrenamiento (*epochs*) y la estrategia de actualización de pesos por lotes (*batch_size*)

```
network.fit(X_train,Y_train,epochs=5,batch_sizes=128,
            validation_data=X_val,Y_val))
```

- Por último se verifica el desempeño de la red neuronal con las métricas fijadas para las pruebas validación ([códigos del libro Francois Chollet](#))

```
network.evaluate(X_test,Y_test)
```

Descenso de gradiente

Algoritmo de optimización para encontrar el mínimo de una función diferenciable.

Los hiperplanos que conforman la entrada de cada neurona, son derivables, y lo que se pretende es optimizar los coeficientes w_{ij} tal que se minimice la función del error E (o función de pérdida), es decir, $\min\{E(y, \hat{y})\}$. En principio es intractable ...

- Sea la función de n valores de entrada a una neurona, esta es una función multivariable:

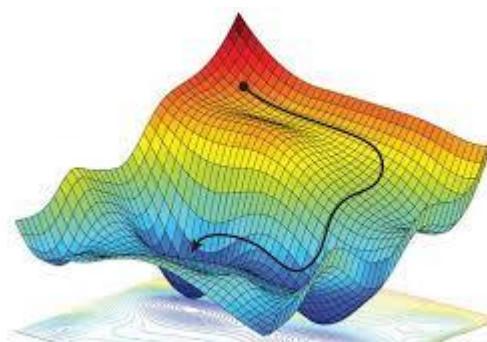
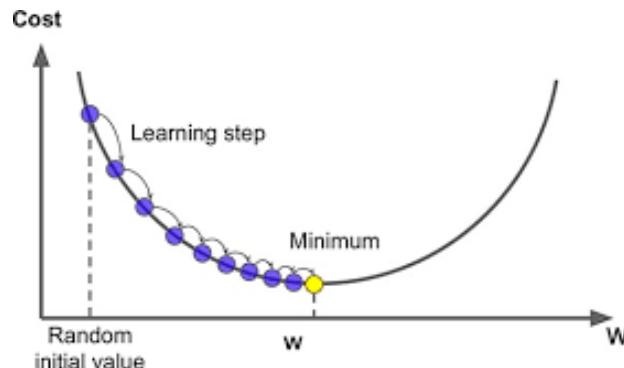
$$y_k = x_1 w_{i_1 j} + x_2 w_{i_2 j} + \dots + x_n w_{i_n j} + b_k$$

La derivada de una función multivariable es el gradiente

$$\nabla y_K = \left(\frac{\partial y_k}{\partial w_{j i_1}}, \frac{\partial y_k}{\partial w_{j i_2}}, \dots, \frac{\partial y_k}{\partial w_{j i_n}} \right)$$

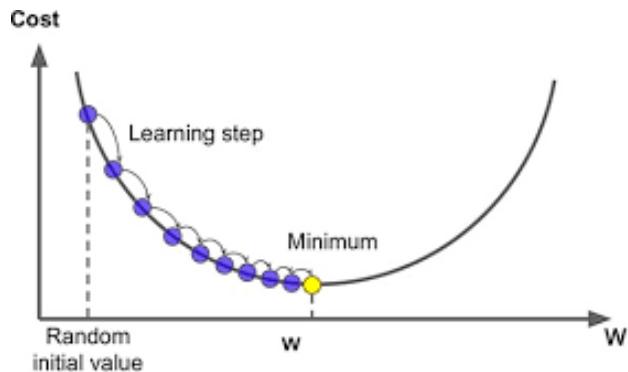
- Entonces el objetivo es minimizar el ajuste de los pesos $w^{k+1} = w^k - \eta \frac{\partial E}{\partial w^k}$

donde η es la tasa de aprendizaje que predetermina la magnitud de los pasos de aproximación al mínimo, al menos local ...



Refinamiento de descenso de gradiente

- La función de error más conocida es el error cuadrático $E = \frac{1}{2}(y - \hat{y})^2$ aunque hay otras funciones de error como la entropía binaria cruzada o el error absoluto medio (regresión)
- Así en la fase forward, se calcula \hat{y} para obtener E y comenzar la fase de retropropagación, por cada peso de la red, hasta la capa de entrada, usando la regla de la cadena. Por ello todos los pesos se ajustarán un pequeño porcentaje, en función a la derivada del error.
- Aunque η no varíe, los pasos de aproximación serán cada vez más pequeños. Lo importante es no tener la tasa de aprendizaje muy grande para evitar la oscilación



Las variantes de descenso de gradiente, conocidas como *optimizer*, permiten una aproximación más eficiente al mínimo

- Hay muchos *optimizer* y entre los más conocidos están: momentum, Nesterov, ADAM, AdamDelta, RMSprop, ... En todos ellos se altera la ecuación de descenso de gradiente

$$w^{k+1} = w^k - \eta \frac{\partial E}{\partial w^k}$$

para lograr más eficiencia en todo el proceso de retropropagación ...

Métodos de optimización para la retropropagación

- El método básico es el descenso estocástico del gradiente (*Stochastic Gradient Descent SGD*) utiliza la fórmula clásica:

$$w_{ij}^{t+1} = w_{ij}^t - \alpha \frac{\partial E}{\partial w_{ij}^t}$$

Sin embargo su convergencia es lenta porque no es posible controlar de manera dinámica la tasa de aprendizaje α .

- La primera variante propuesta ADAGRAD (*Adaptative Gradiente Algorithm*) controla la disminución excesiva de tasa de aprendizaje α e incluye ε para evitar la división por cero

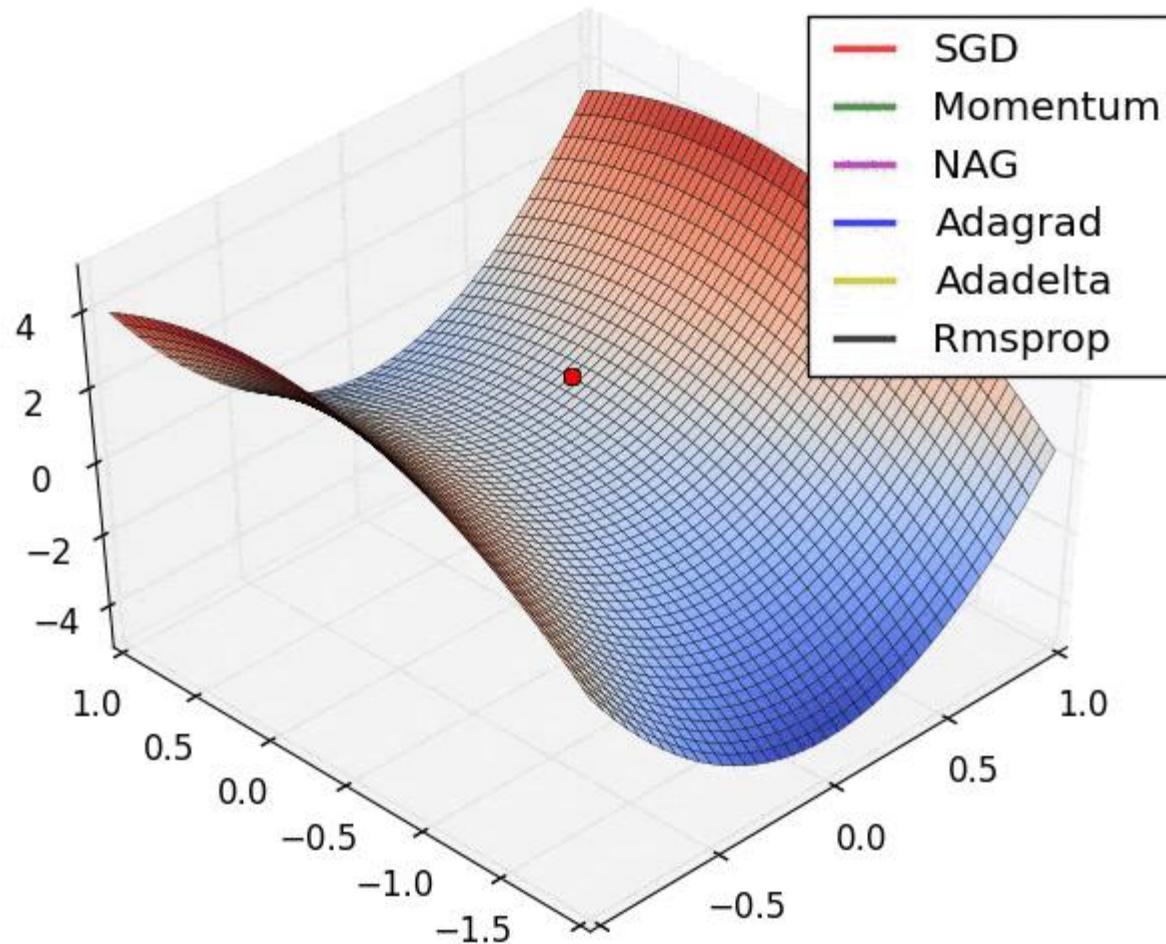
$$C = C + \frac{\partial E}{\partial w_{ij}^t} \quad w_{ij}^{t+1} = w_{ij}^t - \frac{\alpha \frac{\partial E}{\partial w_{ij}^t}}{\sqrt{C + \varepsilon}}$$

- Otra variante muy popular es RMSProp (*Root Mean Square Propagation*) que fue concebida por Geoffrey Hinton pero finalmente no publicó porque lo rechazaron en una conferencia

$$C = C + d.C.(1 - d) \frac{\partial E}{\partial w_{ij}^t} \quad w_{ij}^{t+1} = w_{ij}^t - \frac{\alpha \frac{\partial E}{\partial w_{ij}^t}}{\sqrt{C + \varepsilon}}$$

donde d es la tasa de decaimiento.

Optimizadores



Funciones de pérdida

- La función de pérdida es aquella que se debe minimizar y, para ello, se calcula su gradiente por ser una función multidimensional
- Para las regresiones se usa o el error medio cuadrático (MSE) o el error medio absoluto MAE. Sus representaciones son:

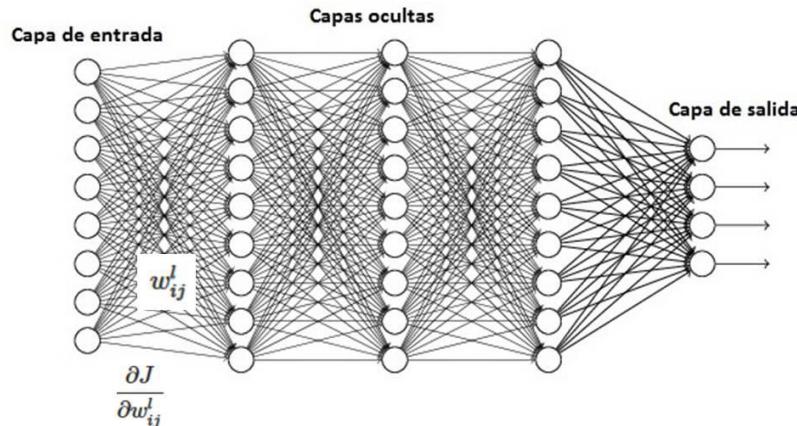
$$MSE = \frac{1}{n} \sum_{j=1}^n (y_i - \hat{y}_i)^2$$

$$MAE = \frac{\sum_{j=1}^n |e_j|}{n}$$

- Para las clasificaciones se usan funciones dependientes de la entropía como en los árboles de clasificación. La más conocida es entropía cruzada. Será categórica si se trata de un clasificador multclases o binaria si es un clasificador a sólo dos clases (categorical_crossentropy o binary_crossentropy) ...

Desvanecimiento y explosión de gradiente

- Cuando los pesos tienen valores menores a uno (1) y muchas capas, el gradiente irá disminuyendo a valores muy pequeños lo que hace ineficaz el ajuste de los pesos (desvanecimiento). Por otro lado, cuando la derivada de la función de activación toma valores muy grandes, ocurre el efecto contrario, los pesos aumentarán exponencialmente y la red no se estabilizará (explosión). Ambos problemas ocurren en redes con muchas capas. El desvanecimiento es conocido como *underfitting*.



Entre las soluciones están: introducir ruido en las funciones de activación de las capas intermedias y asegurarse de que sean no lineales como la activación Leaky ReLU. Para la explosión específicamente se aplica un umbral para evitar el crecimiento del valor en los pesos.

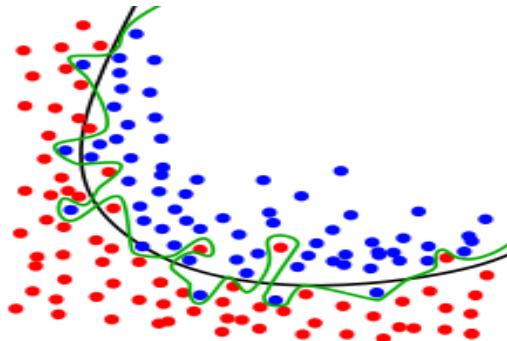
- Otro problema de tener muchas capas y muchos datos de entrenamiento, es la gran cantidad de veces que se deben ajustar los pesos. Una idea es hacer el cálculo de los nuevos pesos para un lote. Es decir, por cada época, definir un grupo de ejemplo, generalmente potencia de 2, para los cuales se hace retropropagación. Un ejemplo en Keras sería:

```
network.fit(X_train, Y_train, epochs=5, batch_size=128)
```

Si se tiene 60.000 ejemplos de entrada, para lotes de 128 por época, entonces se calcularían, sólo 469 corridas de retropropagación por época → 2345 fases de retropropagación en total

Overfitting

- Al igual que en el desvanecimiento de gradiente, el *overfitting* se acrecienta mientras más neuronas y capas tenga la red. Lo sufren muchas técnicas de *machine learning* pero especialmente las redes neuronales profundas



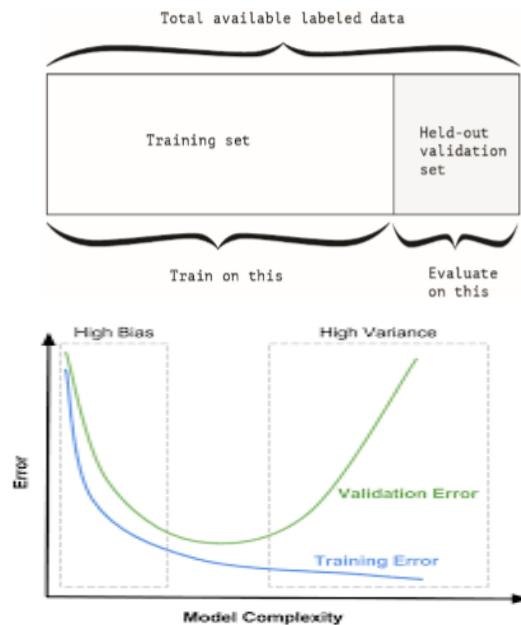
En este caso, clasificar con la línea verde discrimina exactamente dados los casos de entrenamiento pero generaliza mal porque a la llegada de nuevos casos tendrá mayor probabilidad de ser mal clasificados

- En primer lugar, hay que detectar *overfitting* y si, desafortunadamente, la red neuronal sufre de este problema, hay tres estrategias para eliminarlo:
 1. Reducir el tamaño de la red y así disminuir la capacidad de memoria
 2. Agregar factores de regularización al ajuste de los pesos con L-normas
 3. Llevar a cero, aleatoriamente, algunos neuronas de la red, durante la fase *forward*.
- Con respecto, a la detección del *overfitting*, existen métricas para medir la exactitud de la predicción con el conjunto de prueba. Sin embargo, sería más certero con el **conjunto de validación** ya que da un mejor grado de **generalización** porque se evalúa en tiempo de aprendizaje.

Detección de overfitting: conjunto de validación

- Inicialmente se debe tener una aproximación del número adecuado de capas y de neuronas, tasa de aprendizaje, número de épocas, ... los cuales se definen de antemano. Estas variables son conocidas como los **hiperparámetros** (ver [TensorFlow Playground](#)) ...
- **El conjunto de validación** es un hiperparámetro diferente al conjunto de prueba ya que permite ver que tan bien generaliza la red al momento del aprendizaje. Los porcentajes entrenamiento/validación/prueba pueden ser 80/10/10 aunque es variable. La validación tiene tres estrategias:

Simple hold-out validation



```
num_validation_samples = 10000

# Shuffling the data is usually appropriate
np.random.shuffle(data)

# Define the validation set
validation_data = data[:num_validation_samples]
data = [num_validation_samples:]

# Define the training set
training_data = data[:]

# Train a model on the training data
# and evaluate it on the validation data
model = get_model()
model.train(training_data)
validation_score = model.evaluate(validation_data)

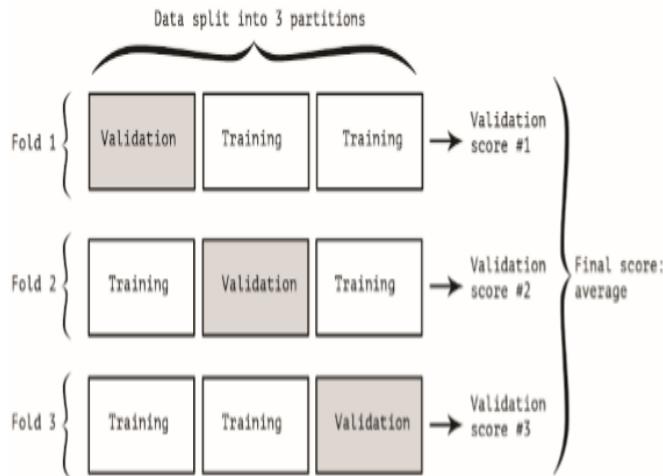
# At this point you can tune your model,
# retrain it, evaluate it, tune it again...

# Once you have tuned your hyperparameters,
# is it common to train your final model from scratch
# on all non-test data available.
model = get_model()
model.train(np.concatenate([training_data,
                           validation_data]))
test_score = model.evaluate(test_data)
```

Conjunto de validación

- Cuando se tienen pocos datos de entrenamiento, el método *simple hold-out validation*, genera mucho sesgo. El conjunto de validación se puede dividir en k particiones y evaluar sobre cada i -esima porción.

K-fold validation



```
k = 4
num_validation_samples = len(data) // k

np.random.shuffle(data)

validation_scores = []
for fold in range(k):
    # Select the validation data partition
    validation_data = data[num_validation_samples * fold: num_validation_samples * (fold + 1)]
    # The remainder of the data is used as training data.
    # Note that the "+" operator below is list concatenation, not summation
    training_data = data[:num_validation_samples * fold] + data[num_validation_samples * (fold + 1):]

    # Create a brand new instance of our model (untrained)
    model = get_model()
    model.train(training_data)
    validation_score = model.evaluate(validation_data)
    validation_scores.append(validation_score)

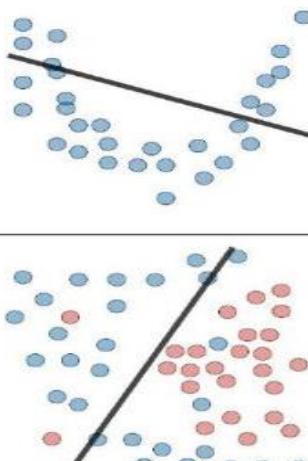
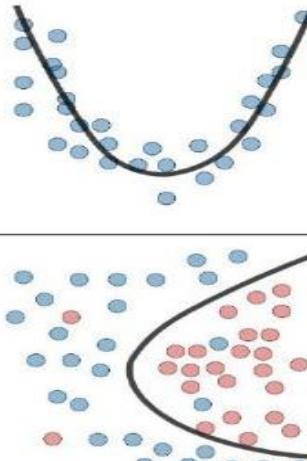
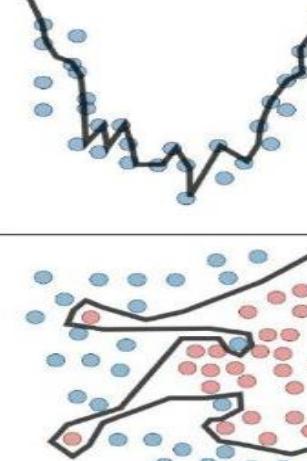
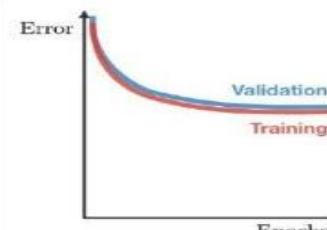
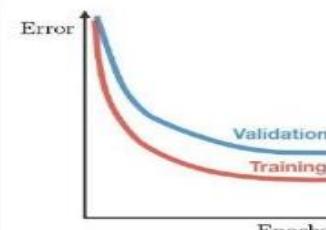
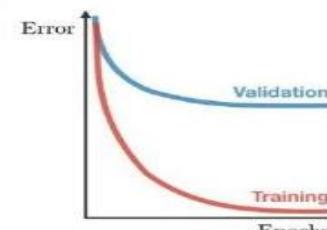
    # This is our validation score:
    # the average of the validation scores of our k folds
    validation_score = np.average(validation_scores)

    # We train our final model on all non-test data available
    model = get_model()
    model.train(data)
    test_score = model.evaluate(test_data)
```

- La tercera estrategia se basa en la anterior pero iterando varias veces para tener una idea del desajuste los hiperparámetros ... Esta estrategia es conocida como: *Iterated K-fold validation*. Es evidentemente más costosa en tiempo de aprendizaje ...
- Cualquiera de estos tres protocolos de evaluación de hiperparámetros debe asegurarse de la representatividad y aleatoriedad de cada uno de los conjuntos (entrenamiento, validación y prueba), evitando los datos repetidos ...

Overfitting vs Underfitting

- Así como se puede sobreaprender también puede ocurrir subaprendizaje. Este último problema también se puede detectar con el conjunto de validación:

| Symptoms | Underfitting | Just right | Overfitting |
|----------------|---|--|--|
| Regression | - High training error - Training error close to test error - High bias | - Training error slightly lower than test error | - Low training error - Training error much lower than test error - High variance |
| Classification |  |  |  |
| Deep learning |  |  |  |
| Remedies | - Complexify model - Add more features - Train longer | | - Regularize - Get more data |

Corregir el overfitting: reducción

- Una vez que se ha detectado el *overfitting*, una primera solución es reducir la talla de la red

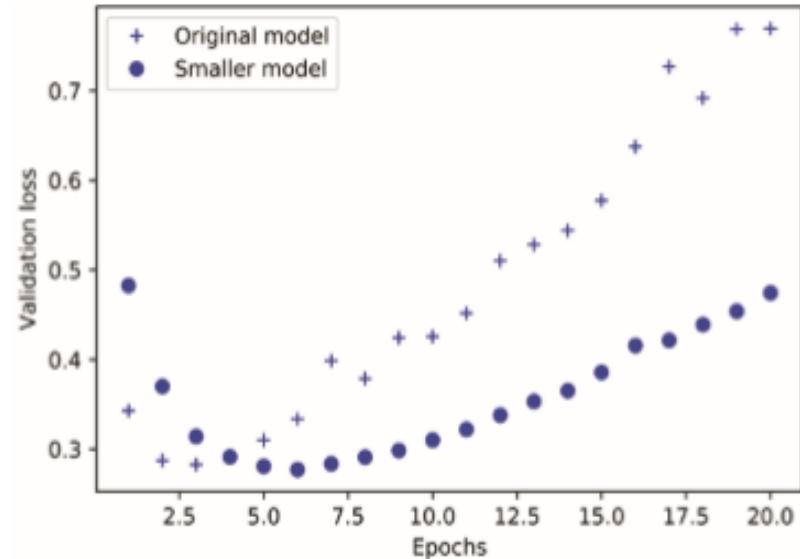
Modelo inicial (+)

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

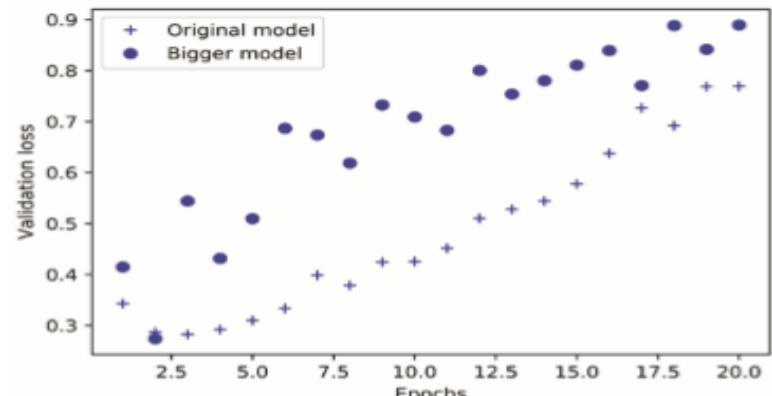
Modelo reducido (•)

```
model = models.Sequential()
model.add(layers.Dense(4, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(4, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```



- Si se agranda el modelo, el *overfitting* empeora y la pérdida es más errática ...

```
model = models.Sequential()
model.add(layers.Dense(512, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```



Corregir overfitting: regularización

- El objetivo es ajustar el error con un factor de castigo para que se altere el cálculo de los pesos.
Por ejemplo, sea el error cuadrático medio para ajustar una red:

$$E = \frac{1}{2}(y - \hat{y})^2 \quad \text{Se agrega el castigo} \quad E = \frac{1}{2}(y - \hat{y})^2 + L_Norma$$

Lo común es agregar la L1-Norma y L2-Norma para las regularizaciones L1 y L2 respectivamente

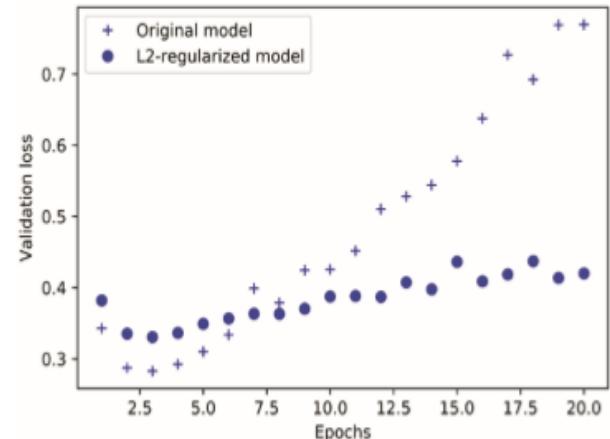
$$L1 = \lambda \sum_{i=1}^n |w_i| \quad L2 = \lambda \sum_{i=1}^n w_i^2 \quad \text{donde } \lambda \text{ es el parámetro de regularización}$$

Visto más general: $E' = \frac{1}{2}(y - \hat{y})^2 + \lambda \sum_{i=1}^n |w_i|$ $E' = \frac{1}{2}(y - \hat{y})^2 + \lambda \sum_{i=1}^n w_i^2$

- En consecuencia al calcular los nuevos pesos con $w^{k+1} = w^k + \eta \frac{\partial E'}{\partial w^k}$ será sobre el error regularizado ...

```
from keras import regularizers

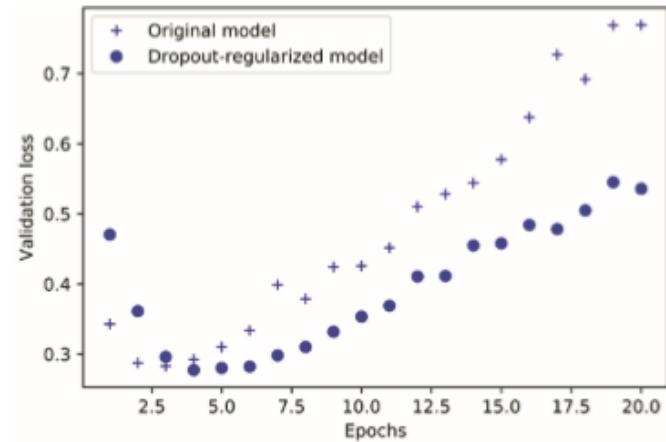
model = models.Sequential()
model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001),
                      activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, kernel_regularizer=regularizers.l2(0.001),
                      activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```



Corregir Overfitting: dropout

- Al igual que la regularización, se aplica durante la fase de entrenamiento. El objetivo es deshabilitar, aleatoriamente, algunas neuronas, en una o varias capas, durante las fases *forward* y *backward*. La tasa de *dropout*, para anular pesos, más común es entre 0.2 y 0.5, es decir, entre 20% a 50% de los pesos de la capa siguiente se llevan a cero.

```
model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1, activation='sigmoid'))
```



- Recapitulando, las tres maneras de corregir el *overfitting* son:

1. Reducir el tamaño de la red neuronal, con ajuste, gracias al conjunto de validación
2. Modificar el cálculo del error de la predicción, mediante L-Normas, para ajustar los pesos durante el *backpropagation*, de tal manera que eviten el *overfitting*.
3. Eliminar neuronas, aleatoriamente, por lo que los pesos en la siguiente capa se desactivan y se reduce la capacidad de memorización de la red neuronal.

Ejemplo en Keras

IMDB es una base de datos de 50,000 registros de texto sobre opiniones de películas donde 50% son reportes positivos y 50% negativos. El texto está transformado en secuencia de enteros que se asocian a palabras

- El primer paso es cargar la base de datos en una matriz de enteros con sólo las 10,000 palabras más frecuentes. Las etiquetas discriminan sólo dos clases: 0 y 1 (crítica positiva o negativa)

```
from keras.datasets import imdb

(train_data, train_labels), (test_data, test_labels) = imdb.load_data(num_words=10000)
```

- Los datos se transforman en una matriz dispersa de 0 y 1 donde las filas representan una crítica y se tienen 10,000 columnas representando la ausencia o presencia de una palabra en esa opinión

```
import numpy as np

def vectorize_sequences(sequences, dimension=10000):
    # Create an all-zero matrix of shape (len(sequences), dimension)
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1. # set specific indices of results[i] to 1s
    return results

# Our vectorized training data
x_train = vectorize_sequences(train_data)
# Our vectorized test data
x_test = vectorize_sequences(test_data)

# Our vectorized labels
y_train = np.asarray(train_labels).astype('float32')
y_test = np.asarray(test_labels).astype('float32')
```

Entrenamiento IMDB

- Se define la red con sigmoide como función de activación de la capa de salida por tratarse de un clasificador binario. Las capas intermedias, como es habitual, tiene activación ReLU.

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

- La función de pérdida o error debe ser la entropía binaria cruzada, por ser un clasificador a dos clases. Se puede hacer de dos maneras:

```
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

```
from keras import losses
from keras import metrics

model.compile(optimizer=optimizers.RMSprop(lr=0.001),
              loss=losses.binary_crossentropy,
              metrics=[metrics.binary_accuracy])
```

- Se define el conjunto de validación y las condiciones de entrenamiento

```
x_val = x_train[:10000]
partial_x_train = x_train[10000:]

y_val = y_train[:10000]
partial_y_train = y_train[10000:]
```

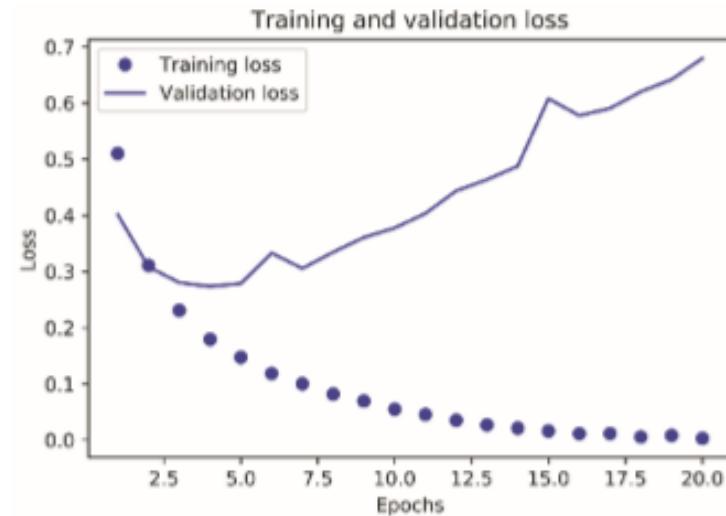
```
history = model.fit(partial_x_train,
                     partial_y_train,
                     epochs=20,
                     batch_size=512,
                     validation_data=(x_val, y_val))
```

Evaluación IMDB

- Primero se visualiza la pérdida para verificar si hay *overfitting* – Si hay !!!

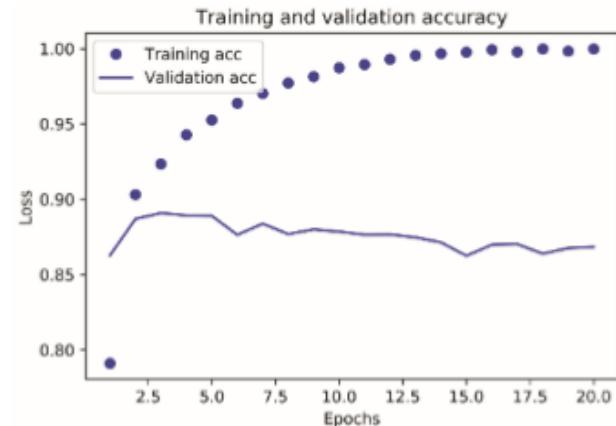
```
>>> history_dict = history.history  
>>> history_dict.keys()  
[u'acc', u'loss', u'val_acc', u'val_loss']
```

```
import matplotlib.pyplot as plt  
  
acc = history.history['acc']  
val_acc = history.history['val_acc']  
loss = history.history['loss']  
val_loss = history.history['val_loss']  
  
epochs = range(1, len(acc) + 1)  
  
# "bo" is for "blue dot"  
plt.plot(epochs, loss, 'bo', label='Training loss')  
# b is for "solid blue line"  
plt.plot(epochs, val_loss, 'b', label='Validation loss')  
plt.title('Training and validation loss')  
plt.xlabel('Epochs')  
plt.ylabel('Loss')  
plt.legend()  
  
plt.show()
```



- Luego la exactitud entre el conjunto de entrenamiento y validación – Mediocre !!!

```
plt.clf()    # clear figure  
acc_values = history_dict['acc']  
val_acc_values = history_dict['val_acc']  
  
plt.plot(epochs, acc, 'bo', label='Training acc')  
plt.plot(epochs, val_acc, 'b', label='Validation acc')  
plt.title('Training and validation accuracy')  
plt.xlabel('Epochs')  
plt.ylabel('Loss')  
plt.legend()  
  
plt.show()
```



IMDB contra el conjunto de prueba

- Se evalúa, después del entrenamiento, con el conjunto de prueba. La exactitud resulta de apenas **88%** ... Los mejores resultados se han logrado con arquitecturas que consiguen un **95%** de *accuracy* ...

```
model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=4, batch_size=512)
results = model.evaluate(x_test, y_test)
```

- Claramente esta es una solución que debe trabajarse más detalladamente con regularización y/o *dropout* para corregir el problema de *overfitting*. [Ejemplo completo del libro](#) y otro [ejemplo simple](#).
- Además para análisis de sentimientos hay otras arquitecturas de redes neuronales que procesan mejor este tipo de información como las redes neurales recurrente, específicamente, LSTM y/o redes neuronales convolucionadas 1D. Un [ejemplo de IMDB](#) que tiene un mejor desempeño justamente con LSTM y CONV 1D.

Pasos previos al entrenamiento

- Preprocesar los datos para cambiar o eliminar atributos sin valor (NaN) mediante un valor por defecto o asignar la media o eliminar el registro, ... También está el problema de los atributos con valores atípicos (*outliers*) que se podría solucionar con normalización o también mediante eliminación del registro ...
- Vectorizar o pasar a matrices multidimensionales, también conocidos como **tensores**. Los datos deben ser valores pequeños entre [0,1] o [-1,1] y homogeneizada
- Una vez que los datos están adaptados a las exigencias de las librerías, se selecciona el tipo de red neuronal y se fijan los hiperparámetros a entonar manualmente (capas y neuronas ...)
- Por último se define el optimizador, la función de activación y pérdida. Estas dos últimas, se deben seleccionar en función al problema que se pretende resolver

| Tipo de problema | Activación última capa | Función de perdida o error |
|---------------------------------------|------------------------|--|
| Clasificación binaria | Sigmoide | binary_crossentropy |
| Clasificación multiclase exclusiva | Softmax | categorical_crossentropy |
| Clasificación multiclase no exclusiva | Sigmoide | binary_crossentropy |
| Regresión para valores arbitrarios | Lineal | Error cuadrático medio (mse) |
| Regresión para valores entre 0 y 1 | Sigmoide | Error cuadrático medio (mse) o binary_crossentropy |

Potencia de Cálculo de una Red Neural Artificial



Funciones Booleanas Combinatorias:

Pueden ser representadas por una red neural a una sola capa intermedia ya que basta con el NAND para obtener cualquier circuito digital combinatorio.



Funciones Continuas:

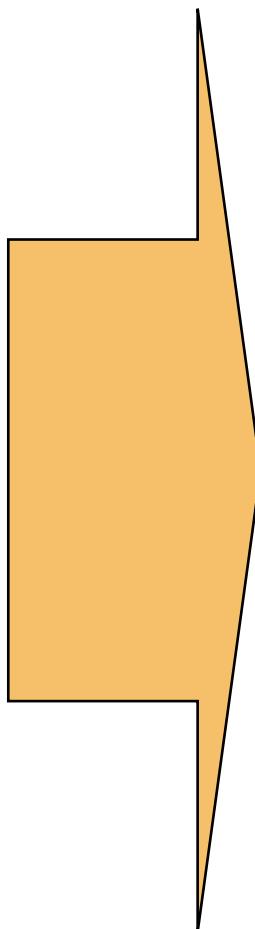
También pueden ser representadas por una red neuronal con al menos dos capas, con precisión arbitraria, a condición que la función de activación, en las capas intermedias sea una función no lineal como ReLU y en la capa de salida lineal sin umbral.



Funciones Arbitrarias n -dimensional:

Cualquier función puede ser aproximada, con precisión arbitraria, con suficientes capas ocultas. Demostrado por el Teorema de Cybenko en 1988. Un teorema más reciente prueba que una red con activación ReLU y al menos $n+1$ capas aproxima cualquier función continua n -dimensional

Consideraciones en una Red Neuronal Artificial



¿ Cuantas unidades o neuronas artificiales ?

Configurable valiéndose del conjunto de validación

¿ Tipo de neurona ?

Depende de la capa (intermedia o salida) y el problema

¿ Topología de la red ?

Tantas capas intermedias como sea posible sin overfitting

¿ Inicialización de los pesos ?

Aleatorio

¿ Número de ejemplos para el entrenamiento ?

Determinado por la cantidad de pesos y cuidando el overfitting

¿ Cómo codificar los datos de entrada y salida ?

Binario es lo común aunque se pueden usar matrices dispersas con valores reales o enteros (aún si se trata texto)

Limitaciones Generales de las Redes Neuronales

- ¿ Cual tipo de red conviene para resolver el problema ?



Se hace empíricamente, en función a problemas parecidos lo cual es muy cuestionable desde el punto de vista formal

- El tiempo de aprendizaje crece exponencialmente por lo que se debe seleccionar adecuadamente el optimizador y considerar indispensable el procesamiento paralelo
- No hay transparencia pues usa enfoque caja negra que impiden saber con certeza como trabaja la red neuron al una vez entrenada



No tienen capacidad de explicación

- Conocimiento a priori no puede ser bien aprovechado en las redes neuronales superficiales. En las redes neuronales profundas es posible dependiendo del tipo de red.