UNIVERSIDAD NACIONAL
DEL NORDESTE

CONICET

I M I T

Módulo: Aprendizaje Profundo

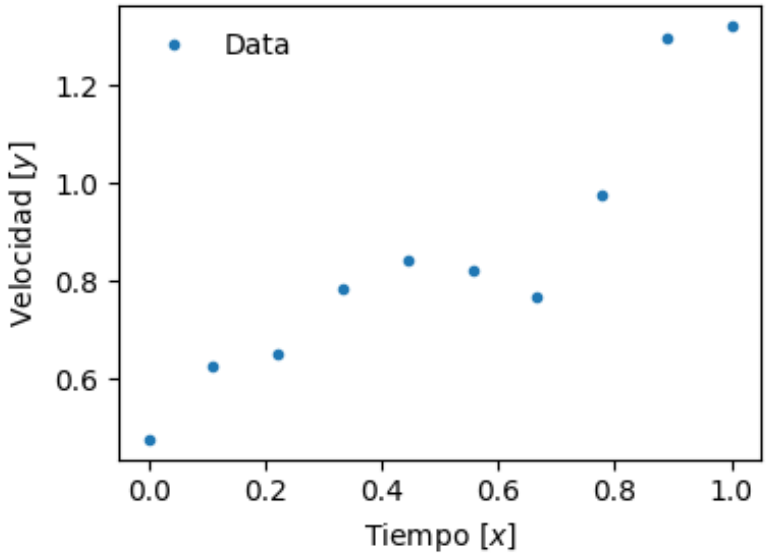
Clase 1: Aprendiendo y prediciendo a partir de datos

Diego Acosta, Emilio Angelina, Leonardo Gomez, Rafael Perez y Manuel Pulido

Planteo: Aprendizaje supervisado

Dado un conjunto de $N_D \gg 1$ pares de datos $\mathcal{D} = \{(x_n, y_n)\}_{n=1}^{N_D}$.

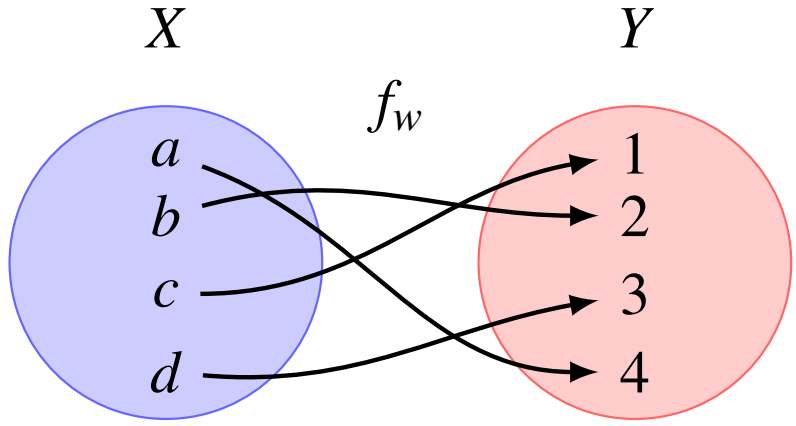
- x_n los llamamos variables de entrada, entradas o “inputs”
- y_n son los “targets”



El objetivo del machine learning es determinar una función o mapa entre el espacio donde yacen los x_n y el de los y_n:

$$y = f_w(x)$$

lo que quiero determinar a partir de los datos son los w de la f_w.



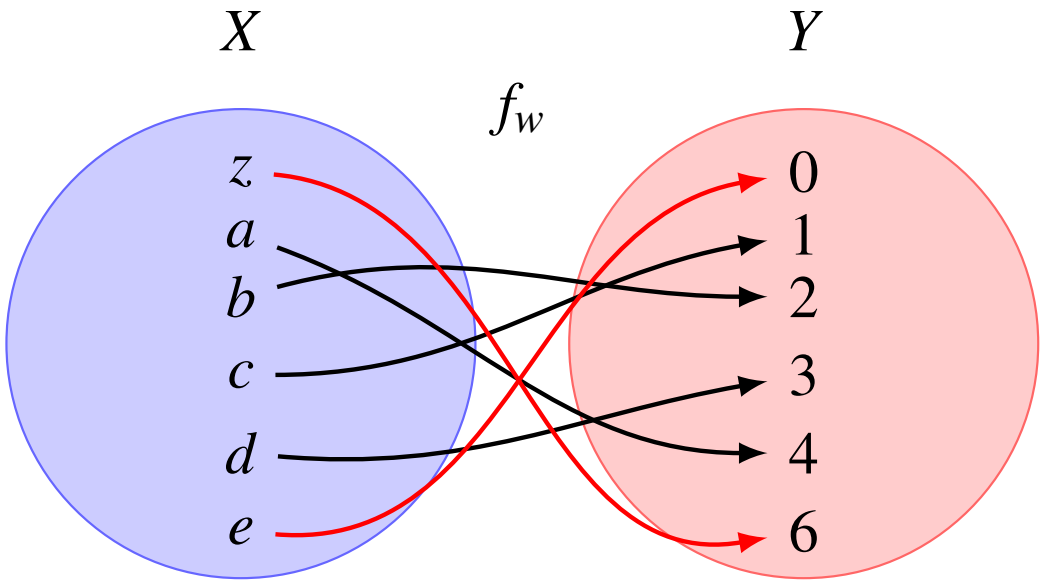
Motivación del aprendizaje supervisado

El modelo permite “generalizar” o “predecir” mas allá de lo aprendido de los datos, es decir si tengo un nuevo x_j y he determinado la f_w, puedo encontrar el valor que le corresponde: $y_j = f_w(x_j)$.

y_j es una predicción del modelo ya que no tenemos ningun dato en x_j.

Ej. $x_5 = e, \quad y_5 = f_w(e) = 0;$

$x_6 = z, \quad f_w(z) = 6$



Las redes neuronales no estudian de memoria, aprender para resolver “nuevos” puntos (nuevos problemas?).

Temario de la clase 1

El marco teórico del aprendizaje profundo

1. Modelos lineales
2. Conexión con la estadística: Estimación por máximo verosimilitud
3. Regularización
4. Redes neuronales: Notación.
5. El largo camino al aprendizaje profundo. Problemas en el entrenamiento.
6. Inestabilidad de los gradientes. Vanishing gradients.

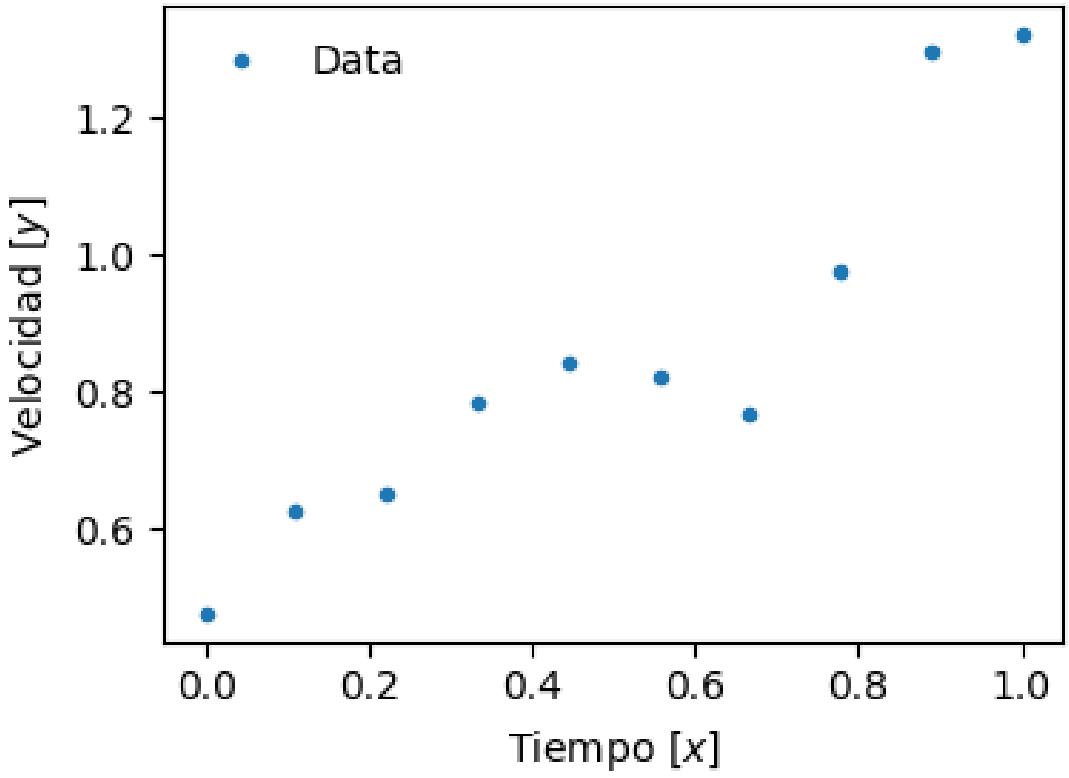
Bibliografía

Para aprender conceptos (y generalizarlos) hay que leer libros no tutoriales:

1. Bishop, C.M. and Bishop, H., 2023. Deep learning: Foundations and concepts. Springer Nature. Será la biblia del deep learning!

2. Prince, S.J., 2023. Understanding deep learning. MIT press.
3. Goodfellow, I, Y Bengio, and A Courville, 2016. Deep Learning. MIT press
4. Zhang, A., Lipton, Z.C., Li, M. and Smola, A.J., 2021. Dive into deep learning. arXiv preprint arXiv:2106.11342.

Ejemplo 1: Determinar la velocidad de una partícula para un dado tiempo



$x \in \mathbb{R}$ y $y \in \mathbb{R}$ entonces queremos encontrar una $f : \mathbb{R} \to \mathbb{R}$

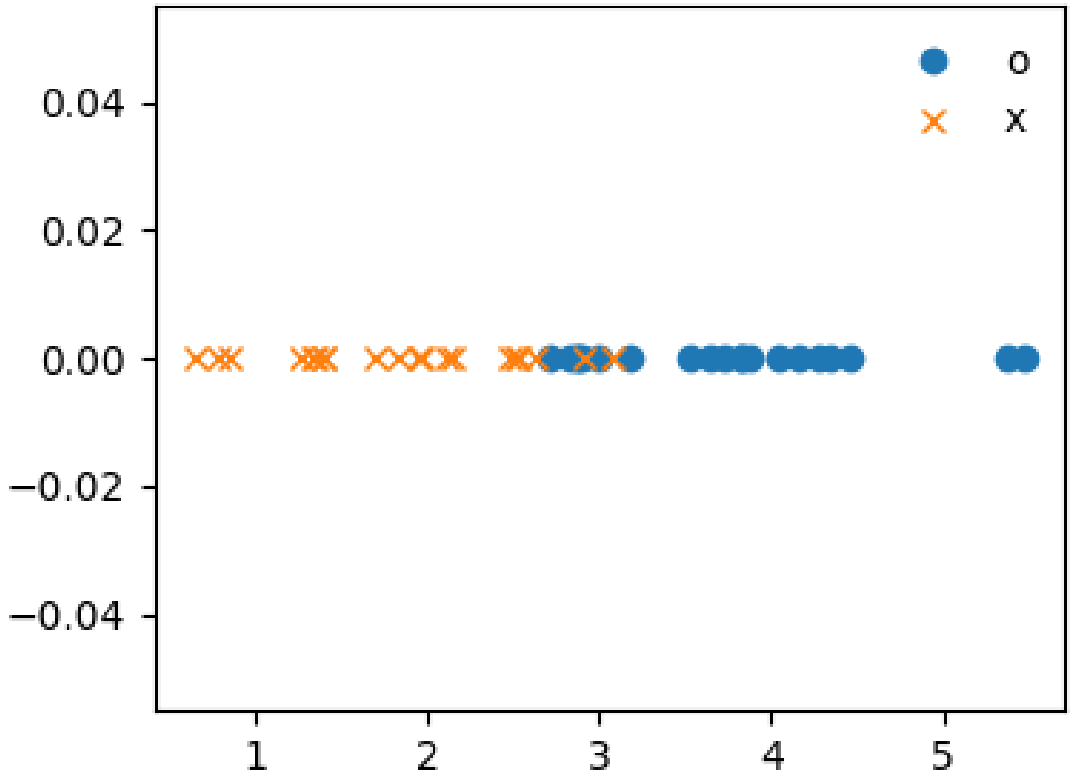
Este es denominado un problema de regresión.

Si la función f se asume apriori es una línea o una recta, $y = w_1 x + w_0$ se denomina regresión lineal.

También se suele llamar método de cuadrados mínimos.

Como asumimos que es una línea recta, debemos determinar los coeficientes w_1 y w_0 que mejor ajustan a los puntos/datos.

Ejemplo 2: Determinar el tipo de pez en base a sus velocidades



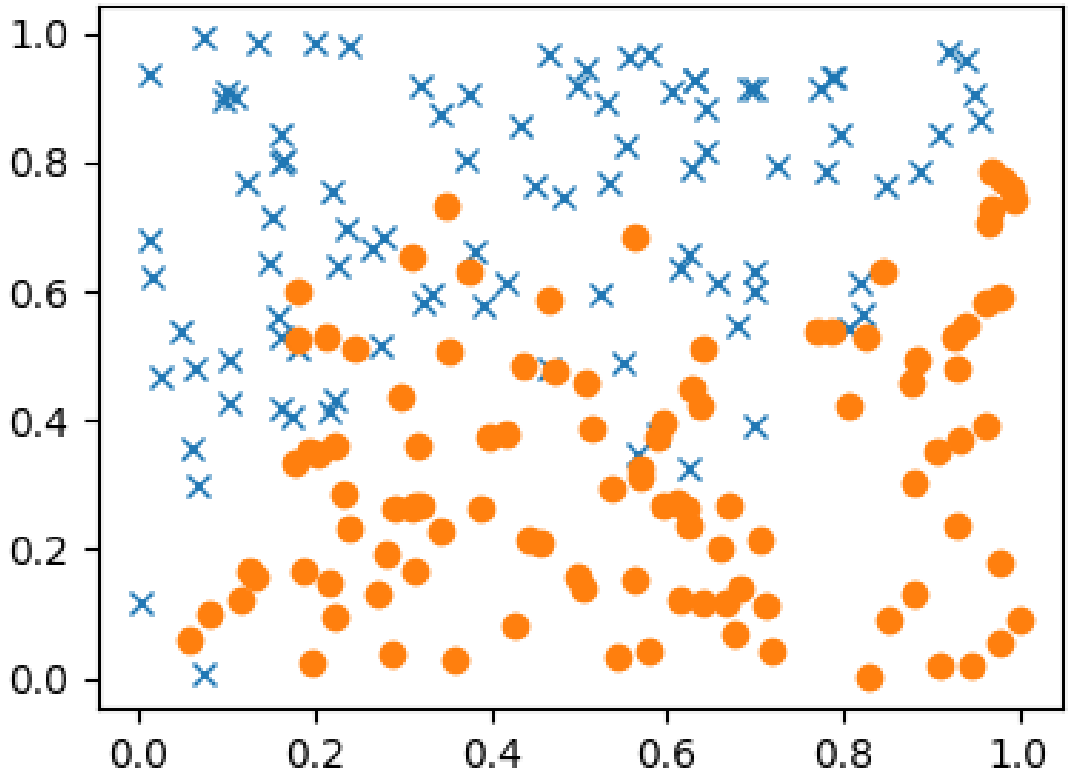
$x \in \mathbb{R}$ y $y \in \{o, x\}$ entonces queremos encontrar una $f : \mathbb{R} \to \{o, x\}$ Los dos tipos o "labels" se los denomina con los numeros binarios $\{0, 1\}$.

Este es denominado un problema de clasificación binaria.

Debemos encontrar donde dividir el dominio, x_D, tal que $x > x_D$ $f(x) =' o'$. Mientras si $x \leq x_D$ $f(x) =' x'$.

¿Cúal les parece que tiene que ser el valor de x_D? Son los datos exactos?

Ejemplo 3: Determinar el tipo de pez en base a sus velocidades y profundidad



$x \in \mathbb{R}^2$ y $y \in \{o, x\}$ entonces queremos encontrar una $f : \mathbb{R}^2 \to \{o, x\}$

Este es denominado un problema de clasificación 2d.

Debemos encontrar donde dividir el dominio, $y(x)$, tal que $y_n > y(x_n)$ $f(x_n) =' \text{o}'$. Mientras si $y_n \leq y(x_n)$ $f(x_n) =' \text{x}'$.

¿Cúal les parece que deba ser la función $y(x)$ que divide? Es una recta? Son los datos exactos? ¿Qué sucede a medida que se aumentan dimensiones?

Función objetivo, de pérdida, de costo

Para aprender vamos a hacer a prueba y error. Mido cuanto erra mi predicción.

La función de pérdida la definimos con alguna métrica que mida las diferencias entre los datos (targets) y las predicciones del modelo.

La Mean Square Error (MSE), distancia euclídea, viene dada por

$$J(\mathbf{w}) = \frac{1}{2N_D} \sum_{n=1}^{N_D} (y_n - f(\mathbf{x}_n, \mathbf{w}))^2$$

Buscamos el conjunto de parámetros $\mathbf{w}$ cuya "función" pase lo mas cerca posible de los datos, produzca el menor error posible de predicción.

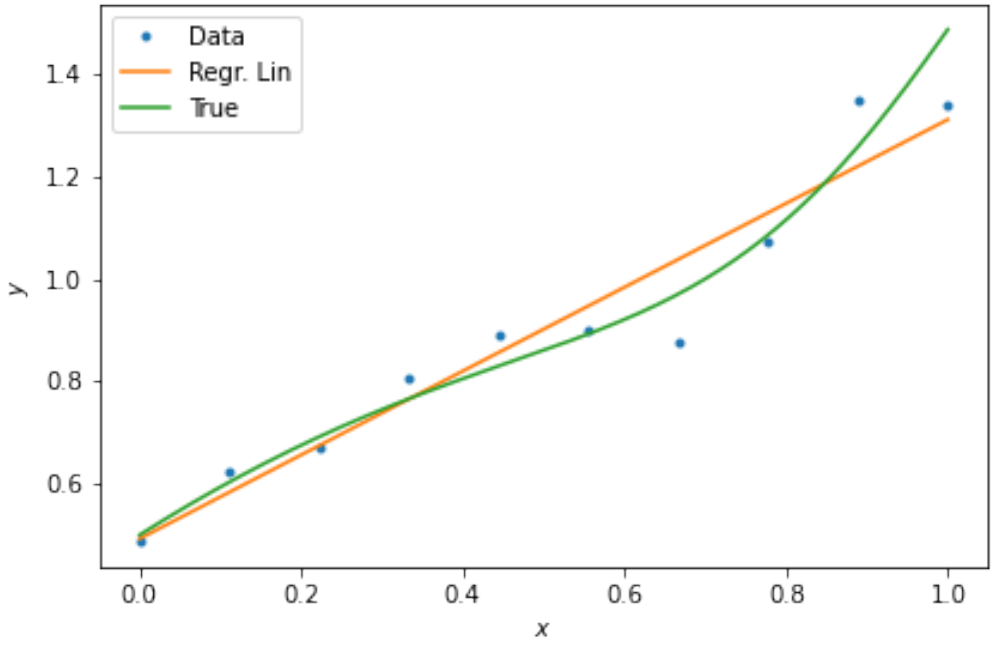
La división por el número de datos (normalización) es conveniente para cuando se quiere comparar J con distintos conjuntos de datos.

Regresión lineal

Definó la design matrix

$$\mathbf{X} = [\mathbf{1}, \mathbf{x}_1, \ldots, \mathbf{x}_{N_x}] \in [N_D, N_x+1]$$

Las columnas de $\mathbf{X}$ son las características. Las filas los datos.



$$J(\mathbf{a}) = \frac{1}{2N_D} (\mathbf{y} - \mathbf{X}\mathbf{w})^\top (\mathbf{y} - \mathbf{X}\mathbf{w})$$

El mínimo viene dado por la raiz del gradiente de J:

$$0 = \nabla_w J(\mathbf{w}) = \frac{1}{N_D} \mathbf{X}^\top (\mathbf{y} - \mathbf{X}\mathbf{w})$$

Los pesos óptimos $\mathbf{w}^*$ vienen dados por:

$$\mathbf{X}^\top \mathbf{X}\mathbf{w} = \mathbf{X}^\top \mathbf{y} \qquad \rightarrow \qquad \mathbf{w}^* = \left(\mathbf{X}^\top \mathbf{X}\right)^{-1} \mathbf{X}^\top \mathbf{y}$$

Inversión de matriz. $\mathbf{X}^\top \mathbf{X}$ Puede ser singular. Se puede hacer con SVD.

Generalizacion a modelos lineales

Cualquier superposición de un conjunto ortogonal de funciones es lineal con respecto a los parámetros (independientemente de las dependencias en las inputs):

$$y^f = \sum_{j=1}^{N_c} w_j \phi_j(\mathbf{x})$$

donde $\{\phi_1(\mathbf{x}), \phi_2(\mathbf{x}), \cdots, \phi_{N_C}(\mathbf{x})\}$ es un conjunto ortogonal de funciones.
N_c es la capacidad o la complejidad del modelo (número de funciones base).

Si hay múltiples outputs las puedo trabajar por separado. A menos que . . .

Design matrix para modelos lineales generalizados

Puedo trabajar con la design matrix

$$\mathbf{X} = [\phi_k(\mathbf{x}_n)]_{nk}$$

Las columnas de $\mathbf{X}$ son las características/features $\mathbf{\Phi}_k$. Las filas los datos.

$$\mathbf{X} = \begin{pmatrix} \phi_1(\mathbf{x}_1) & \phi_2(\mathbf{x}_1) & \cdots & \phi_{N_C}(\mathbf{x}_1) \\ \phi_1(\mathbf{x}_2) & \phi_2(\mathbf{x}_2) & \cdots & \phi_{N_C}(\mathbf{x}_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_1(\mathbf{x}_{N_D}) & \phi_2(\mathbf{x}_{N_D}) & \cdots & \phi_{N_C}(\mathbf{x}_{N_D}) \end{pmatrix}$$

Los pesos óptimos vienen (de nuevo) dados por:

$$\mathbf{w}^* = \left(\mathbf{X}^\top \mathbf{X}\right)^{-1} \mathbf{X}^\top \mathbf{y}$$

Para regresión polinomial: $\mathbf{\Phi}(x) = [1, x, x^2, x^3, \cdots, x^{N_c - 1}]$ $\mathbf{X} \in [N_D, N_c]$
¿Qué sucede si el input es de 2 variables? y de 3 variables? ¿cómo aumenta el número de coeficientes?

Ejemplos de funciones base

- Funciones polinómicas: $\phi_j(\mathbf{x}) = x^j$. Puedo incluir el sesgo: $\phi_0(\mathbf{x}) = 1$
- Otros ejemplos: senos y cosenos, polinomios de Legendre, etc.$\rightarrow$ Problemas son funciones globales cambios en una región llevan a cambios en otras.

  Locales:
- Funciones base radiales $\phi_j(\mathbf{x}) = \exp\left[-\frac{(x-\mu_j)^2}{2\sigma^2}\right]$ donde μ_j define la ubicación y σ la escala de la función (ligadas a los datos). Se suele definir el parámetro: $\alpha = \sigma^{-2}/2$ denominado bandwidth o precisión.
- Base de funciones con localización en el espacio y en los números de ondas: Ondeletas.

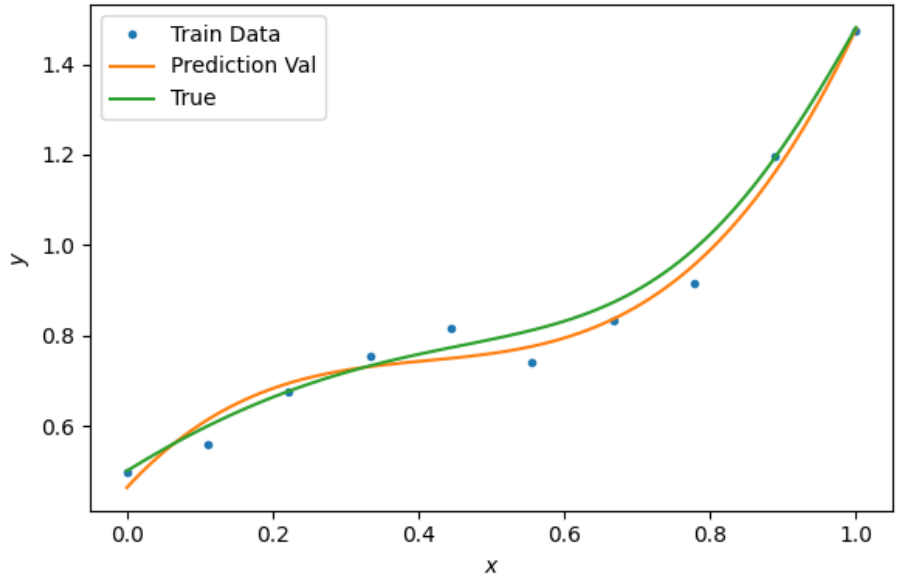
Definición de un modelo en pytorch para regresión polinomial

```
class poli_model(torch.nn.Module):
    def __init__(self,n_Degree,n_outputs):
    " Defino el modelo lineal. Grado del polinomio "
        super().__init__()
        self.n_inputs=n_Degree+1
        self.lm = torch.nn.Linear(n_inputs,n_outputs,bias=True)

    def forward(self, x):
        " Agrego como features los terminos del polinomio de grado
                                                n_inputs-1 "
        x_aug=[x[:,0]**i for i in range(1,self.n_inputs)]
        x_aug=torch.stack(x_aug)
        return self.lm(x_aug.T)
```

Ajuste polinómico

Dentro de la clase poli_model agrego dos funciones:

```
def design(self,x):
  x_aug=[x[:,0]**i for i in range(
                    self.n_inputs)]
  x_aug=torch.stack(x_aug)
  return x_aug.T

def optweights(self,x,y):
  with torch.no_grad(): # no
                    training
    X=self.design(x)
    minv=torch.linalg.inv(X.T @ X)
    w= minv @ (X.T @ y)
    self.lm.bias[0]=w[0]
    self.lm.weight[0,:]=w[1:,0]
```



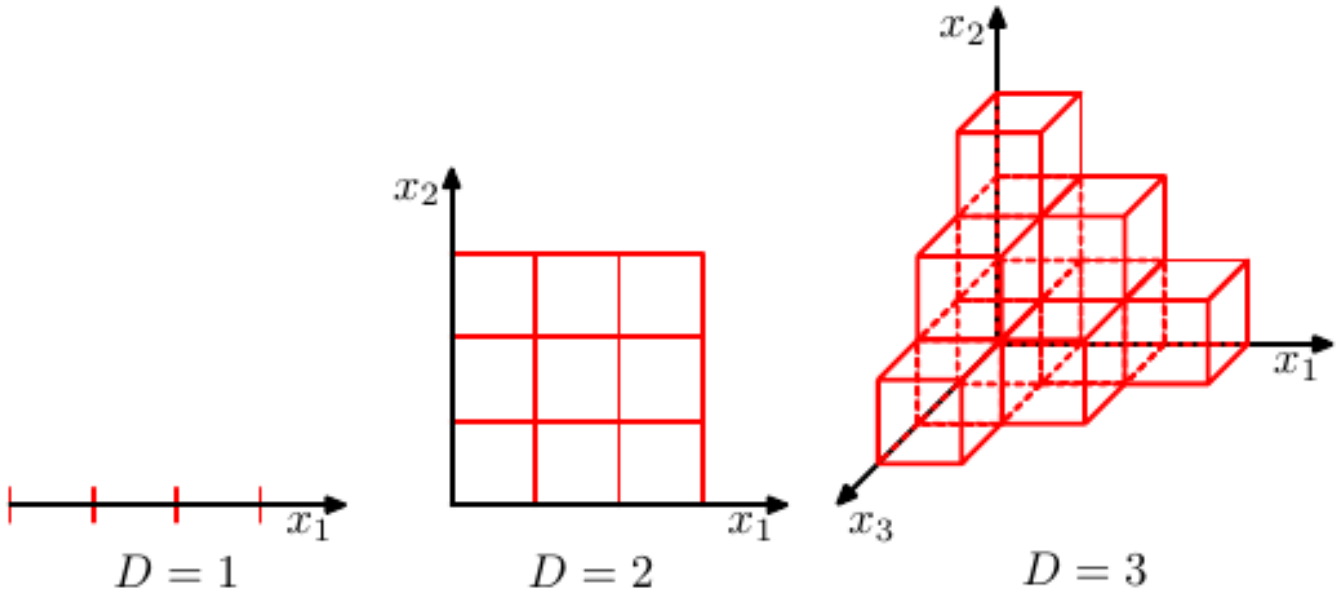
Polinomio óptimo.

```
model=poli_model(3,1)
model.optweights(x_train,y_train
                 )
with torch.no_grad(): # no
                    training
  y_pred_val=model(x_val)
```

Curso de la dimensionalidad

La base de funciones a utilizar se define a priori, antes de que se conozcan los datos.

Esto trae aparejado el "curso de la dimensionalidad": debo aumentar exponencialmente el numero de funciones base cuando aumento la dimensionalidad del espacio de entrada.



Funciones base adaptativas

Los datos generalmente no son independientes estan correlacionados entre sí y por lo tanto viven en un subespacio (manifold) de menor dimensión.

Las redes neuronales tienen:

- funciones base adaptativas. Las variaciones de las funciones de base (definidas a través de los parámetros de la red) son aquellas que corresponden al manifold de los datos
- Dentro del manifold, las variables target varían en algunas direcciones preferenciales. Las redes capturan estas direcciones en el espacio de entrada a través de las funciones base.

Patil, D.J., Hunt, B.R., Kalnay, E., Yorke, J.A. and Ott, E., 2001. Local low dimensionality of atmospheric dynamics. *Physical Review Letters*, **86**, p.5878.

Interpretación estadística de la optimización

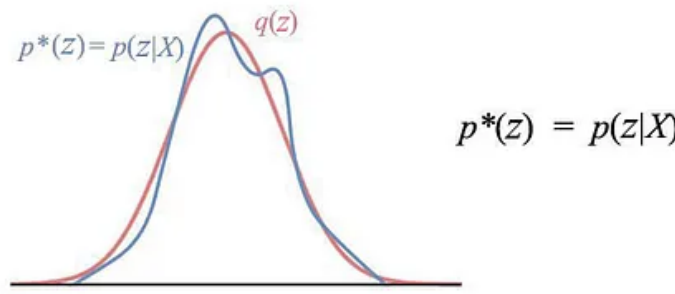
Si tenemos datos de un problema supervisado, $\mathcal{D} = \{(\mathbf{x}_n, \mathbf{y}_n)\}_{n=1}^{N_D}$, estos son muestras de una distribución de probabilidad,

Datos: $p_{\mathcal{D}}(\mathbf{y}|\mathbf{x})$

Esta es desconocida y es el objetivo de machine learning determinarla. Supongamos nuestro modelo probabilístico de predicción se define por una familia de densidades parametrizadas a través de $\mathbf{w}$,

Modelo: $p_f(\mathbf{y}|\mathbf{x}, \mathbf{w})$

La idea es encontrar el conjunto de parámetros que acerque lo mas posible la densidad de predicción a la densidad de los datos,



Diferencias entre la distribución de los datos y la del modelo propuesto

La divergencia de Kullback-Leibler $\mathcal{D}_{KL}(p_{\mathcal{D}}\|p_f)$ nos "mide" la diferencias entre dos distribuciones:

$$\mathcal{D}_{KL}(p_{\mathcal{D}}\|p_f) \doteq \int p_{\mathcal{D}}(y|\mathbf{x}) \log\left(\frac{p_{\mathcal{D}}(y|\mathbf{x})}{p_f(y|\mathbf{x},\mathbf{w})}\right) \mathrm{d}\mathbf{x}$$

Aprovecho la asimetría en la definición de la $\mathcal{D}_{KL}$ (p_f es la distribución que no conozco).

Usando los datos:

$$\mathcal{D}_{KL}(p_{\mathcal{D}}\|p_f|) \doteq \mathbb{E}_{p_{\mathcal{D}}}\left[\log\left(\frac{p_{\mathcal{D}}(y|\mathbf{x})}{p_f(y|\mathbf{x},\mathbf{w})}\right)\right] = -\sum_{n=1}^{N_D} \log p_f(y_n|\mathbf{x}_n,\mathbf{w}) + C$$

Estoy haciendo Monte Carlo con la integral.

Puedo interpretar a la distribución *en la* integral como:

$p_{\mathcal{D}}(y|\mathbf{x}) = \sum_n \delta(x - x_n)$

Máximo verosimilitud

Si los datos son asumidos iid la función verosimilitud, $\mathbf{w}$ y $\mathbf{\Sigma}$ determinan la media y la covarianza de la distribución

$$p(\mathbf{y}^{(1)}, \cdots, \mathbf{y}^{(N_D)} | \mathbf{x}^{(1)}, \cdots, \mathbf{x}^{(N_D)}; \mathbf{w}, \mathbf{\Sigma}) = \prod_{n=1}^{N_D} p(\mathbf{y}^{(n)} | \mathbf{x}^{(n)}; \mathbf{w}, \mathbf{\Sigma})$$

Fuerte hipótesis de ML: Los datos son iid. Contraejemplos: Cambio climático. Sistemas no-autónomos.

El logaritmo de la verosimilitud viene dado por

$$l(\mathbf{w}, \mathbf{\Sigma}) = \log p(\mathbf{y}^{(1:N_D)} | \mathbf{x}^{(1:N_D)}; \mathbf{w}, \mathbf{\Sigma}) = \sum_{n=1}^{N_D} \log p(\mathbf{y}^{(n)} | \mathbf{x}^{(n)}; \mathbf{w}, \mathbf{\Sigma})$$

Hemos obtenido el negativo de la divergencia de KL.

Maximizar la log-verosimilitud es equivalente a minimizar la DKL.

Máximo verosimilitud para distribuciones Gaussianas

Hipótesis Gaussiana: $p(y^{(n)}|\mathbf{x}^{(n)};\mathbf{w},\Sigma) = \mathcal{N}(y_n|f(\mathbf{x}_n,\mathbf{w}),\Sigma)$

$$l(\mathbf{w},\boldsymbol{\Sigma}) = C + \frac{N}{2}\log|\boldsymbol{\Sigma}| - \sum_{n=1}^{N_D}[\mathbf{y}_n - f(\mathbf{x}_n,\mathbf{w})]^\top \boldsymbol{\Sigma}^{-1}[\mathbf{y}_n - f(\mathbf{x}_n,\mathbf{w})]$$

Para $\boldsymbol{\Sigma} = \mathbf{I}$,

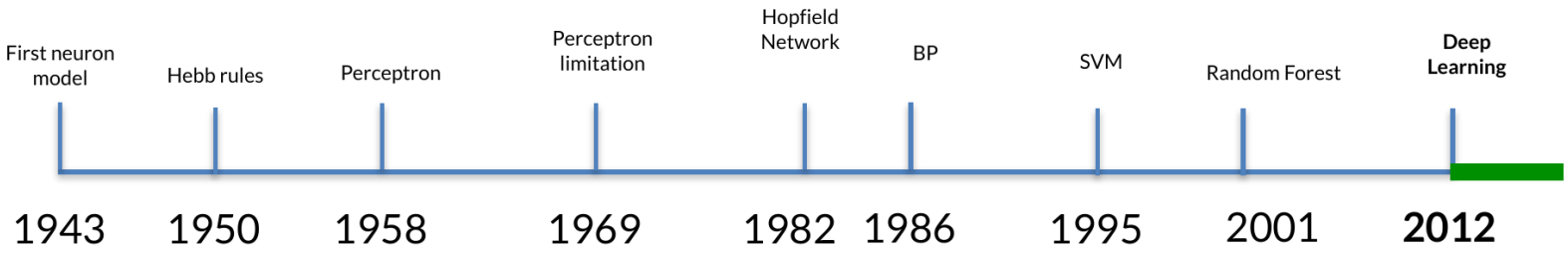
$$J(\mathbf{w}) = C - \sum_{n=1}^{N_D}[\mathbf{y}_n - f(\mathbf{x}_n,\mathbf{w})]^\top [\mathbf{y}_n - f(\mathbf{x}_n,\mathbf{w})]$$

Hemos obtenido la MSE

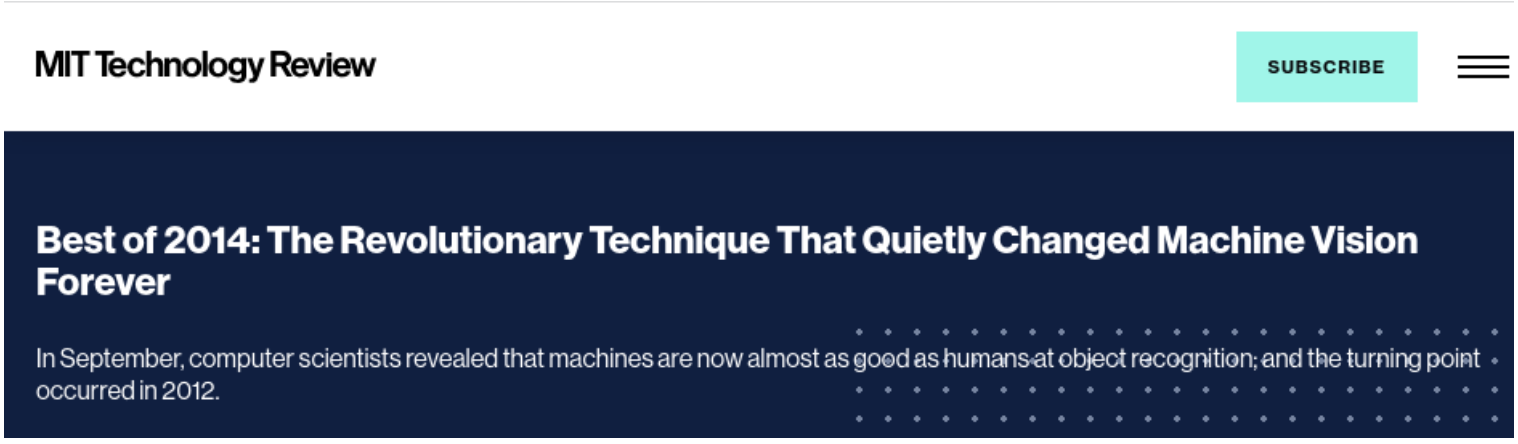
Cuando minizamos la función de pérdida J:

- Estamos maximizando la log-verosimilitud con respecto a los parámetros.
- Estamos buscando la densidad que mejor ajusta los datos (en una flia).

Historias de las redes neuronales

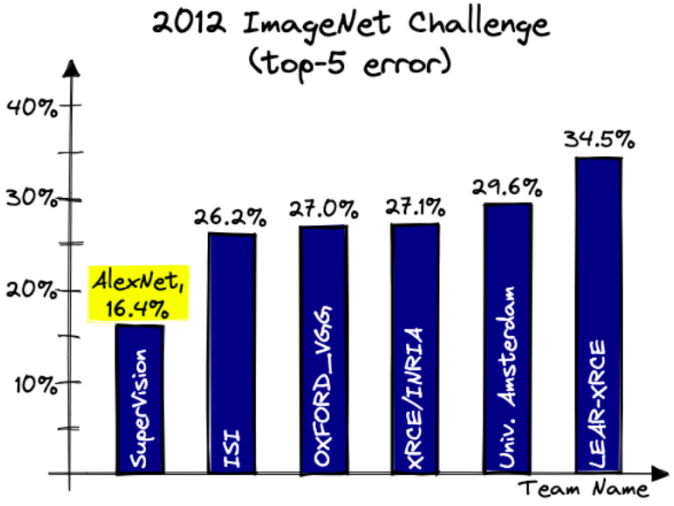


El hito de la revolución



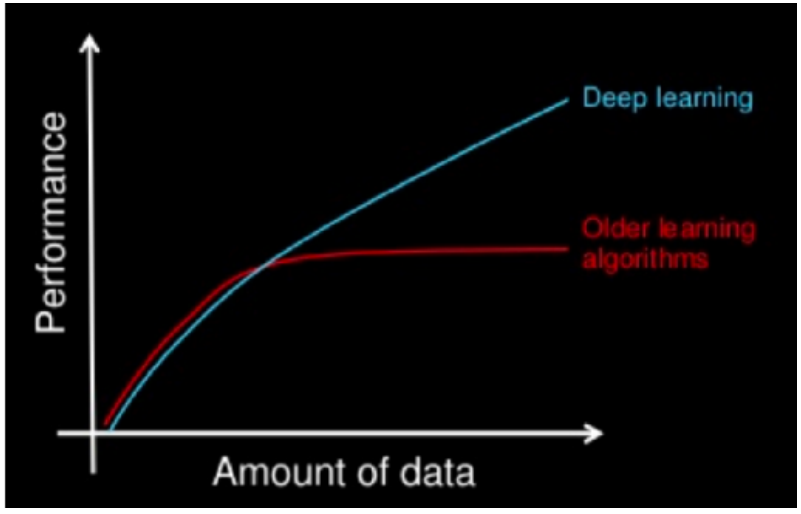
MIT news

Inicio de la revolución.

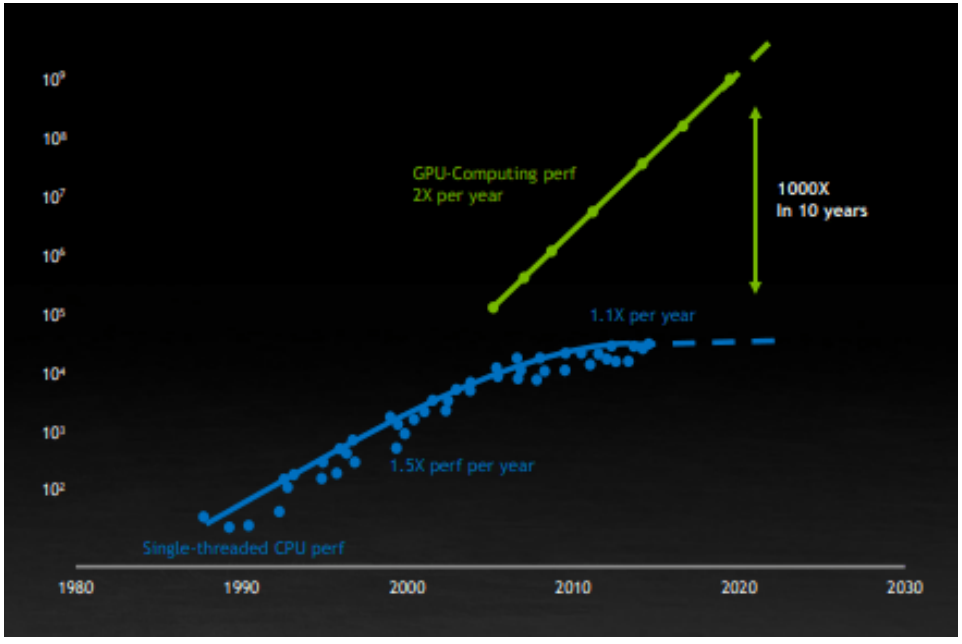


Krizhevsky A, Sutskever I, Hinton GE, 2012 Imagenet classification with deep convolutional neural networks. 120.000 Citas!!!

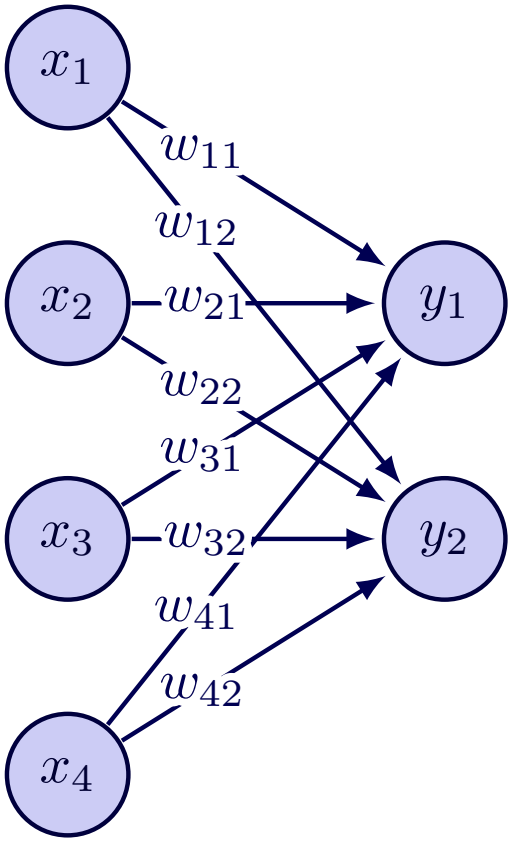
Causas de la revolución



Fuente Seeking alpha.



Red neuronal lineal. (bah un producto matricial) .



- Cada nodo de entrada se conecta con todos los nodos/variables de salida. Fully connected.
- Los pesos son los parámetros de la red.
- Pueden pensarse como links/canales de comunicación y de acuerdo al peso los nodos van a estar mas o menos comunicados/relacionados.
- Si $w_{ij} = 0$ significa que el nodo x_i no esta comunicado/relacionado con el y_j.

$$y_j = \sum_{i=0}^{N_x} w_{ij} x_i = \mathbf{w}_j \cdot \mathbf{x}$$

Es un producto matricial: $\mathbf{y} = \mathbf{W}\,\mathbf{x}$

Agregando no-linealidad

Todo lo que hacemos es aplicar una función no-lineal h a la salida de las redes lineales ya definidas:

$$\mathbf{y} = h(\mathbf{W}\,\mathbf{x})$$

Se denomina función de activación. (Ejemplos: relu, silu, sigmoide, tanh, sin, etc)
La h tiene por objeto subdividir el dominio de la red lineal.
Deja pasar la información para una región. Elimina la entrada para el resto del dominio.

Sucesiva aplicación de capas: composición

Aplicación de los pesos y la función de de activación de la primera capa:

$$\mathbf{x}^{(1)} = h^{(0)}\left(\mathbf{W}^{(0)}\mathbf{x}^{(0)}\right)$$

Segunda capa:

$$\mathbf{x}^{(2)} = h^{(1)}\left(\mathbf{W}^{(1)}\mathbf{x}^{(1)}\right)$$

$$\mathbf{x}^{(2)} = h^{(1)}\left(\mathbf{W}^{(1)}h^{(0)}\left(\mathbf{W}^{(0)}\mathbf{x}^{(0)}\right)\right)$$

Las distintas capas son una composición de funciones sobre las anteriores.

Magia de las redes de las redes neuronales

Si h es nolineal, las redes neuronales con múltiples capas son aproximadores universales.

Hornik, K., Stinchcombe, M. and White, H., 1989. Multilayer feedforward networks are universal approximators. Neural networks, 2, 359-366.

- Las redes neuronales con múltiples capas son capaces de aproximar cualquier función a cualquier grado de precisión requerido.
- Esto no dice nada del número de neuronas ni del entrenamiento

Notación para describir una red

Los componentes de la red fully connected son:

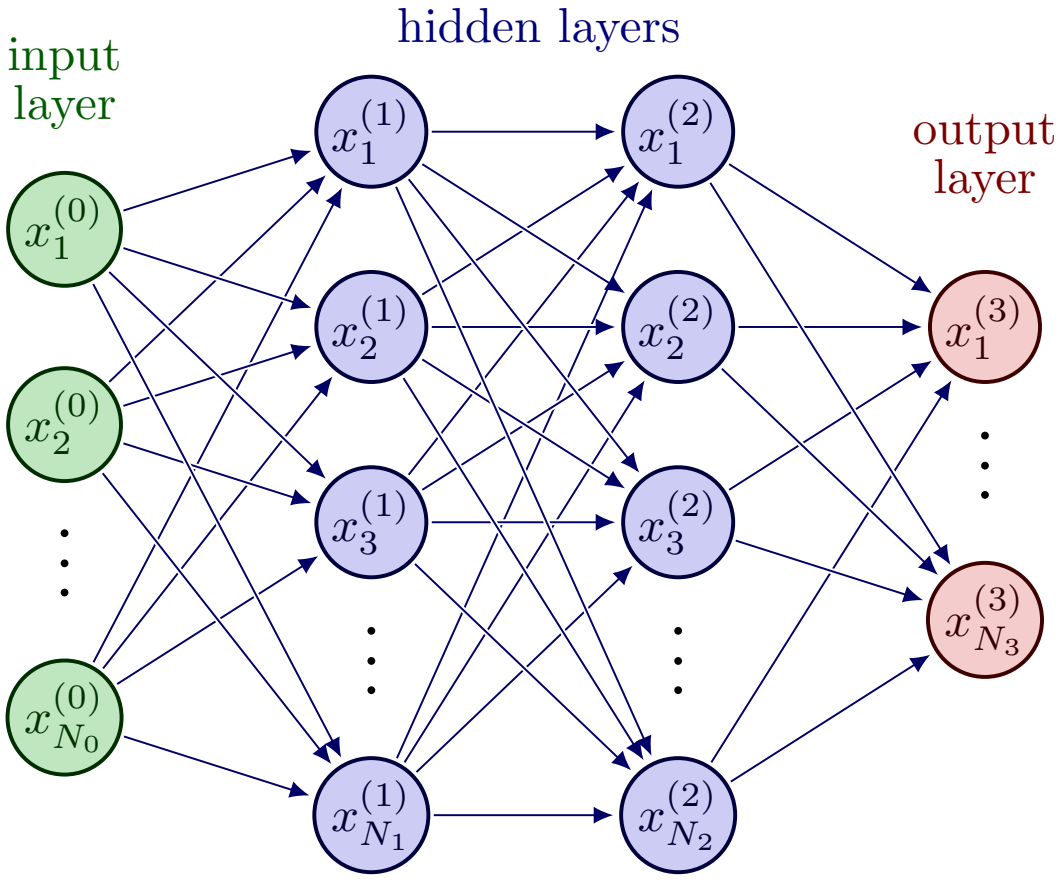
- Variables $x_j^{(k)}$, representan el nodo/neurona j-esimo de la capa k-esima
- Pesos $w_{ij}^{(k)}$ conexión entre nodo i de la capa k con el j de la capa $k+1$
- N_k cantidad de neuronas/nodos en la capa k.
- $h^{(k)}$ función de activación de la capa k-ésima

Una variable arbitraria de la capa k viene dada por los valores de la capa anterior:

$$x_j^{(k+1)} = h^{(k)}\left(\sum_{i=0}^{N_k} w_{ij}^{(k)} x_i^{(k)}\right)$$

Defino $x_0^{(k)} = 1$ asi que $w_{0j}^{(k)}$ es el sesgo.

Red Fully Connected (FC)



- Agregamos capas múltiples a la red que ya teníamos.
- Supra-índices número de la capa.
- Para cada capa debemos hacer la operación:

$$x_j^{(k+1)} = h^{(k)}\left(\sum_{i=0}^{N_k} w_{ij}^{(k)} x_i^{(k)}\right)$$

$$x_j^{(K)} = h^{(K-1)}\left(h^{(K-2)}\left(\cdots\left(h^{(0)}\left(\sum_{i=0}^{N_k} w_{ij}^{(k)} x_i^{(k)}\right)\right)\cdots\right)\right)$$

Defino una red FC en pytorch

```
import torch, torch.nn as nn
class FCNNmodel(nn.Module):
    " Genera una Fully connected NN con funciones de activacion "
    def __init__(self,layers,activation_fn,activation_output=None):
        super().__init__()
        modules = []
        for i in range(len(layers)-2):
            modules.append(nn.Linear(layers[i], layers[i+1]))
            modules.append(activation_fn)
        modules.append(nn.Linear(layers[len(layers)-2],
                                 layers[len(layers)-1])) #output
        if activation_output is not None:
            modules.append(activation_output)
        self.NNmdl = nn.Sequential(*modules)
    def forward(self,x):
        return self.NNmdl(x)
if __name__=="__main__":
     Nlayers=[5,16,16,5]
     Net=FCNNmodel(Nlayers,nn.ReLU())
     x=torch.randn(Nlayers[0])
     y=Net(x)
```

Propagación hacia adelante

Capa k-esima:

$$\mathbf{x}^{(k)} = h^{(k-1)}\left(\mathbf{W}^{(k-1)} h^{(k-2)}\left(\dots h^{(0)}\left(\mathbf{W}^{(0)}\mathbf{x}^{(0)}\right)\dots\right)\right)$$

Las k capas representan k composiciones de funciones.

Input de la red: $\mathbf{x}^{(0)} = \mathbf{x}$, output $\mathbf{x}^{(K)} \doteq \mathbf{y}$

Propagación hacia adelante:

$\mathbf{x}^{(0)} \to \mathbf{x}^{(1)} \dots \to \mathbf{x}^{(K)}$

Input $\to$ Capa 2 $\to$ Capa 3 $\to$ Output $\to$ Función de pérdida

$J(\text{Input}, \text{Output})$

$$J(\mathbf{W}) = \frac{1}{2N_d}\sum_{n=1}^{N_d}\left\|\mathbf{x}_n^{(K)} - \mathbf{y}_n\right\|^2$$

$$= \frac{1}{2N_d}\sum_{n=1}^{N_d}\left\|h^{(K-1)}\left(\mathbf{W}^{(K-1)} h^{(K-2)}\left(\dots h^{(0)}\left(\mathbf{W}^{(0)}\mathbf{x}_n^{(0)}\right)\dots\right)\right) - \mathbf{y}_n\right\|^2$$

Entrenamiento

Para cualquier modelo nolineal o red neuronal con funciones de activacion nolineales es imposible determinar los parámetros analíticamente.

“Training the network involves searching in the weight space of the network/model for a value of $\mathbf{w}$ that produces a function that fits the provided training data well” (MacKay, 2003).

Gradiente de la función de pérdida de una red

NATURE VOL. 323 9 OCTOBER 1986 — LETTERS TO NATURE — 533

Learning representations by back-propagating errors

David E. Rumelhart*, Geoffrey E. Hinton†
& Ronald J. Williams*

* Institute for Cognitive Science, C-015, University of California, San Diego, La Jolla, California 92093, USA
† Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Philadelphia 15213, USA

We describe a new learning procedure, back-propagation, for networks of neurone-like units. The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. As a result of the weight adjustments, internal 'hidden' units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units. The ability to create useful new features distinguishes back-propagation from earlier, simpler methods such as the perceptron-convergence procedure[1].

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986) Learning representations by back-propagating errors, Nature, 323, 533-536.

Backpropagation: Gradientes en pytorch

autograd Torch nos calcula el backprop en forma automatica!

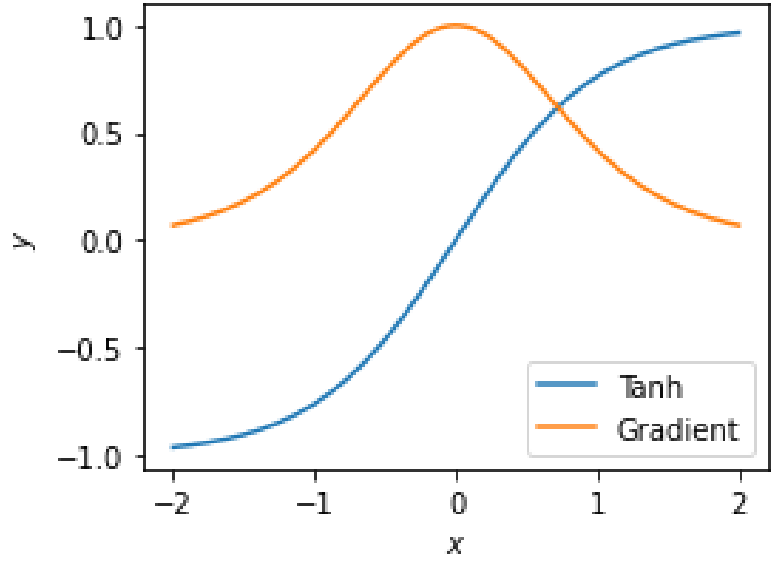
Para que autograd realice la diferenciación automática se debe especificar:

```
In [38]: x=torch.linspace(-2.,2.,200,requires_grad=True)
```

Todas las operaciones a partir de x van a requerir de backpropagación:

```
In [39]: h=torch.nn.Tanh()
In [40]: y=h(x)
In [41]: print(y)
tensor([ -0...], grad_fn=<TanhBackward0>)
```

```
In [42]: J=y.sum()
In [43]: J.backward()
In [44]: print(x.grad[5])
```

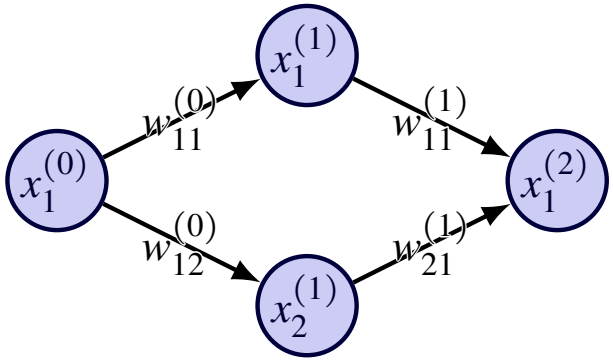


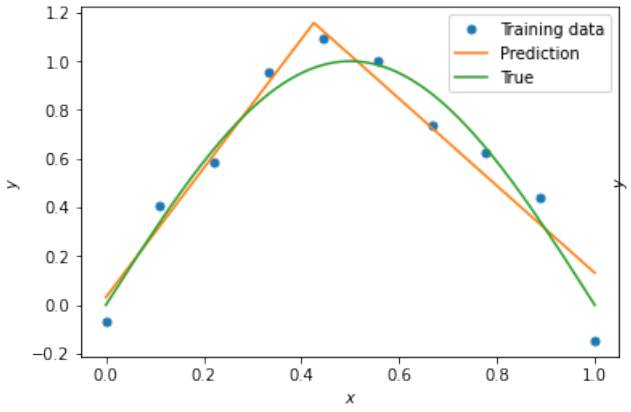
Guarda para cada variable dependiente el tipo de operación. Guarda estados. Con el `.backward()` calcula cada gradiente.

Funciones de activación. Interpretación de las redes neuronales

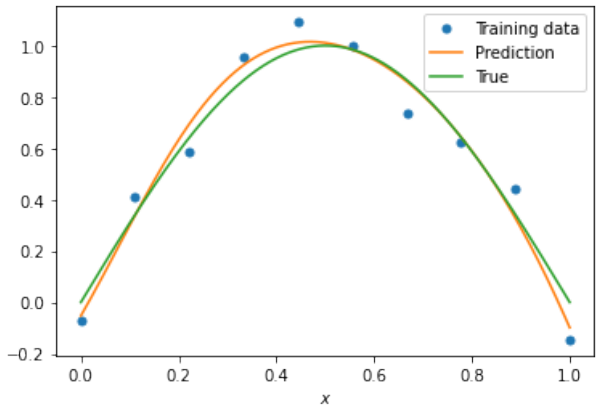
Red ultra sencilla 1-2-1 (2 neuronas internas).

Funciones de activación en ambos enlaces.

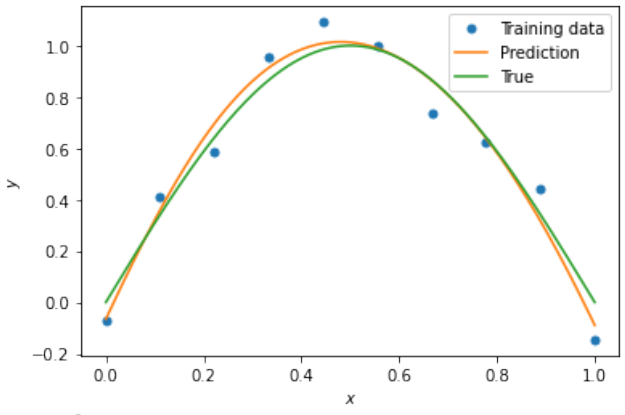




ReLU



Tanh



Sigmoid

¿Que puede decir de los parámetros w?

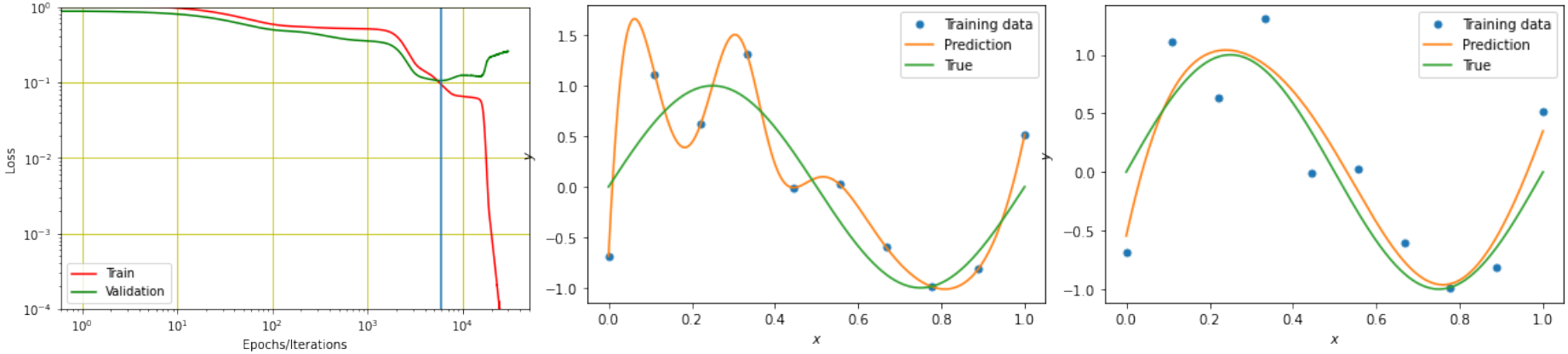
¿Que roles tienen las funciones de activación en cada capa?

Reglas para la selección de funciones de activación

- Capas internas. Se seleccionan de acuerdo a la arquitectura:
  - Siempre comience por una ReLU.
  - CNN: ReLU
  - Redes Recursivas: Tanh o Sigmoide
  - Redes muy profundas (>40): Swish
- Capa de salida (output layer). Se seleccionan de acuerdo al problema:
  - Regresión: Función lineal (sin función de activación/identidad)
  - Clasificación binaria: Sigmoide/Logística
  - Clasificación multi-clase: Softmax
  - Clasificación multi-etiqueta: Sigmoide

Overfitting y sobre-entrenamiento

Modelo 10 neuronas internas. Fn activación: Tanh.

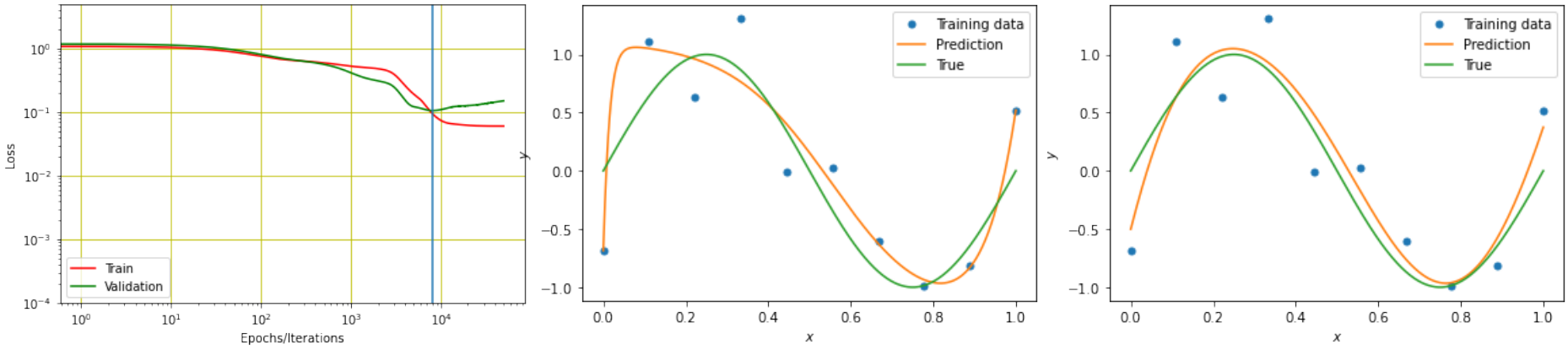


Demasiada complejidad del modelo.

- Fuerte ajuste de los datos de entrenamiento (Izq) pero sin poder de generalización
- Detengo el entrenamiento cuando la J de validación empieza a crecer (Der).

¿Que sucede si elegimos la capacidad correcta?

Modelo de 3 neuronas internas (capacidad correcta)



¿Porqué hay un “leve” overfitting?

- Muy similar performance al modelo mas complejo. Misma capacidad de generalización.
- Der detengo en J validación mínima.

Early stopping

- Free lunch (de Hinton/Bengio): single training realization enough.
- Evaluar después de cada época la función de pérdida en el conjunto de validación.
- Guardar los parámetros con mejor error de validación.
- Detener el entrenamiento cuando aumentamos las epocas y la función de pérdida de validación no disminuye
- Paciencia: Cuidado con la estocasticidad (no detener si son unas pocas épocas de crecimiento).

Marco Bayesiano

Limitaciones de la estimación por verosimilitud: No nos dice nada sobre la incerteza de la predicción. Asume total desconocimiento a priori de los parámetros.

¿Que pasa si asumimos que tenemos un conocimiento apriori de los parámetros expresado en la densidad de probabilidad a priori,

$$p(\mathbf{w}|\mathcal{D}) = \frac{p(\mathcal{D}|\mathbf{w})p(\mathbf{w})}{p(\mathcal{D})}$$

$$p(\mathcal{D}|\mathbf{w}) = \prod_{n=1}^{N_D} p(y_n|\mathbf{w}, \mathbf{x}_n)p(\mathbf{x}_n)$$

Asumiendo Gaussianidad: $p(y|\mathbf{w}, \mathbf{x}) = \mathcal{N}(y|f(\mathbf{x}; \mathbf{w}), \sigma^2)$

$$p(y|\mathbf{w}, \mathbf{x}) = (2\pi\sigma^2)^{-1/2} \exp\left(-\frac{1}{2\sigma^2}[y - f(\mathbf{x}; \mathbf{w})]^2\right)$$

Misma función verosimilitud.

Redes Bayesianas. Aprenden la incerteza de los parámetros

Regularización. Weight decay

Queremos castigar en la función de costo los valores altos de los parámetros (favoreciendo los pequeños)

$$J(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N_D}\left(y^{(n)} - f(\mathbf{x}^{(n)};\mathbf{w}\right)^2 + \frac{\lambda}{2}|\mathbf{w}|^q$$

- Concepto útil cuando trabajamos con pocos datos y modelos complejos (redes profundas).
- Mas utilizadas L2, $q = 2$ y L1, $q = 1$ (lasso).

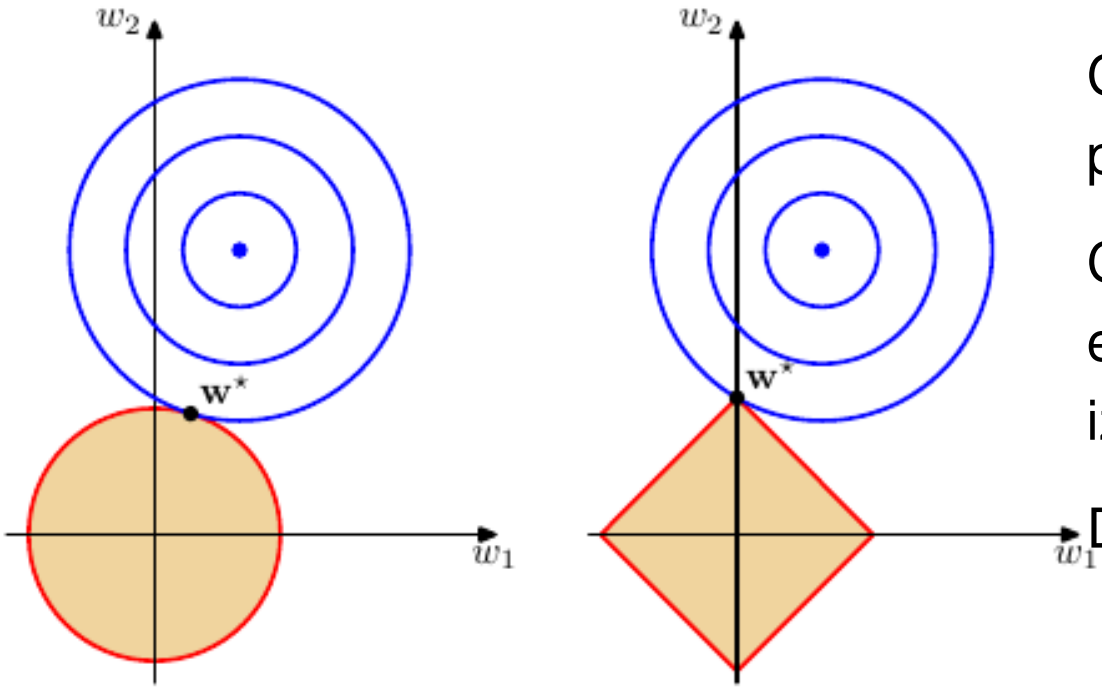
Weight decay: Se cambian directamente los parámetros en el paso de update (no la J).

$$\mathbf{w} = \hat{\mathbf{w}} - \eta\nabla_{\mathbf{w}}J\Big|_{\hat{\mathbf{w}}} - \eta\lambda\mathbf{w}$$

Esta implementado en pytorch: `optimizer = tor.optim.Adam( NNmdl.parameters(), weight_decay=0.01)`

Regularización lasso. Parámetros esparsos

La regularización L1, se conoce como lasso, produce soluciones esparsas.

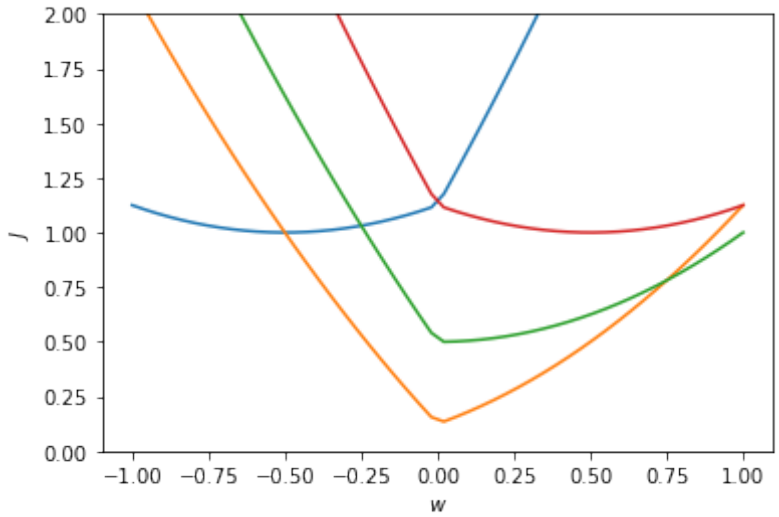


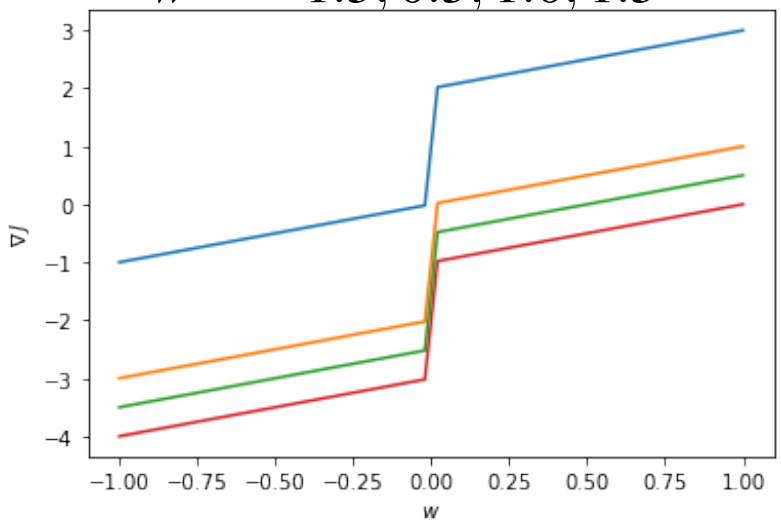
Contornos azules función de pérdida sin regularización.

Contornos rojos restricciones establecidas por la regularización.

De Bishop 1996.

$\hat{w} = -1.5, 0.5, 1.0, 1.5$





Interpretación estadística de la regularización

La regularización L2 (ridge regression):

$$J(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N_D}\left(y^{(n)} - f(\mathbf{x}^{(n)};\mathbf{w}\right)^2 + \frac{\lambda}{2}\mathbf{w}^\top\mathbf{w}$$

Renombrando constantes, puede reinterpretarse por el logaritmo de Gaussianas:

$$J(\mathbf{w}) = -\log\left\{\exp\left[-\sum_{n=1}^{N_D}\frac{\Sigma^{-1}}{2}\left(y^{(n)} - f(\mathbf{x}^{(n)};\mathbf{w}\right)^2\right]\exp\left[-\sum_{n=1}^{N_D}\frac{B^{-1}}{2}\mathbf{w}^\top\mathbf{w}\right]\right\}$$

Esto puede escribirse como productos de Gaussianas:

$$J(\mathbf{w}) = -\log\left(\prod_{n=1}^{N_D} p(y^{(n)}|\mathbf{x}^{(n)};\mathbf{w},\Sigma)p(\mathbf{w})\right) = -\log\left(\frac{p(\mathbf{w}|\mathbf{X},\mathbf{y})}{p(\mathbf{y})}\right)$$

Entonces la regularización puede ser interpretada como una densidad a-priori de los parámetros asumiendo tienen media 0.

En lugar de MaxLik estamos haciendo MAP!

Extensiones de la función de pérdida

La función de pérdida general de Mikonski la definimos funcionalmente con

$$L_q(\mathbf{x}, y) = |f(\mathbf{x}) - y|^q$$

Con $q = 2$, norma L2, tenemos el MSE. $q = 1$ es la norma L1.

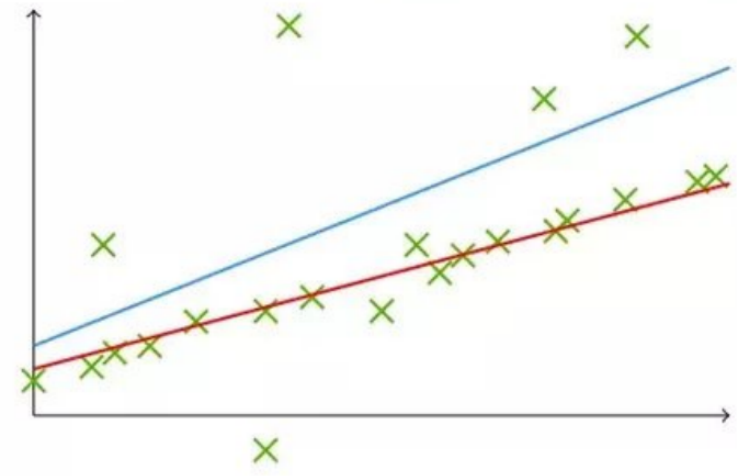
- L_1 distribución de Laplace
- L_2 normal

Obteniendo el valor esperado de la densidad conjunta,

$$J = \mathbb{E}(L_q) \doteq \iint |f(\mathbf{x}) - y|^q p(\mathbf{x}, y) \mathrm{d}\mathbf{x}\mathrm{d}y$$

Dados $\left\{\mathbf{x}^{(n)}, t^{(n)}\right\}_{n=1}^{N_D}$, la integral de Monte Carlo de J nos termina dando

$$J = \sum_{n=1}^{N_D} |f(\mathbf{x}^{(n)}) - y^{(n)}|^q$$



L_1 rojo. L_2 celeste.

En forma comparativa $L2$ le da mayor peso a los outliers.

¿Qué paso entre 1986 y 2000's?

Rta. Desierto (con unas "pocas" excepciones: 1989 universal aprox, 1997 LSTM, 1998 OCR cartas código postal cartas).

El problema de gradiente explotando o desapareciendo

Las redes profundas tienen aparejado un problema:

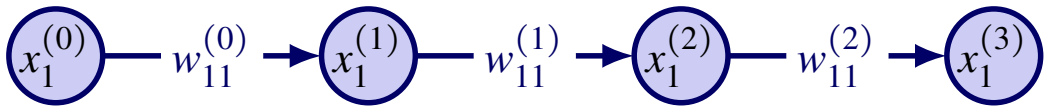
- Los gradientes a lo largo de las capas se van haciendo muy pequeños (vanishing). Se pierde la sensibilidad a los parámetros.
- Los gradientes pueden explotar creciendo desproporcionadamente.

Si el producto de las derivadas del peso y de la función de activación en un nodo exceden el valor de uno estos van a continuar creciendo exponencialmente a lo largo del camino.

$$x_j^{(k+1)} = h^{(k)} \left(\sum_{i=0}^{N_k} w_{ij}^{(k)} x_i^{(k)} \right)$$

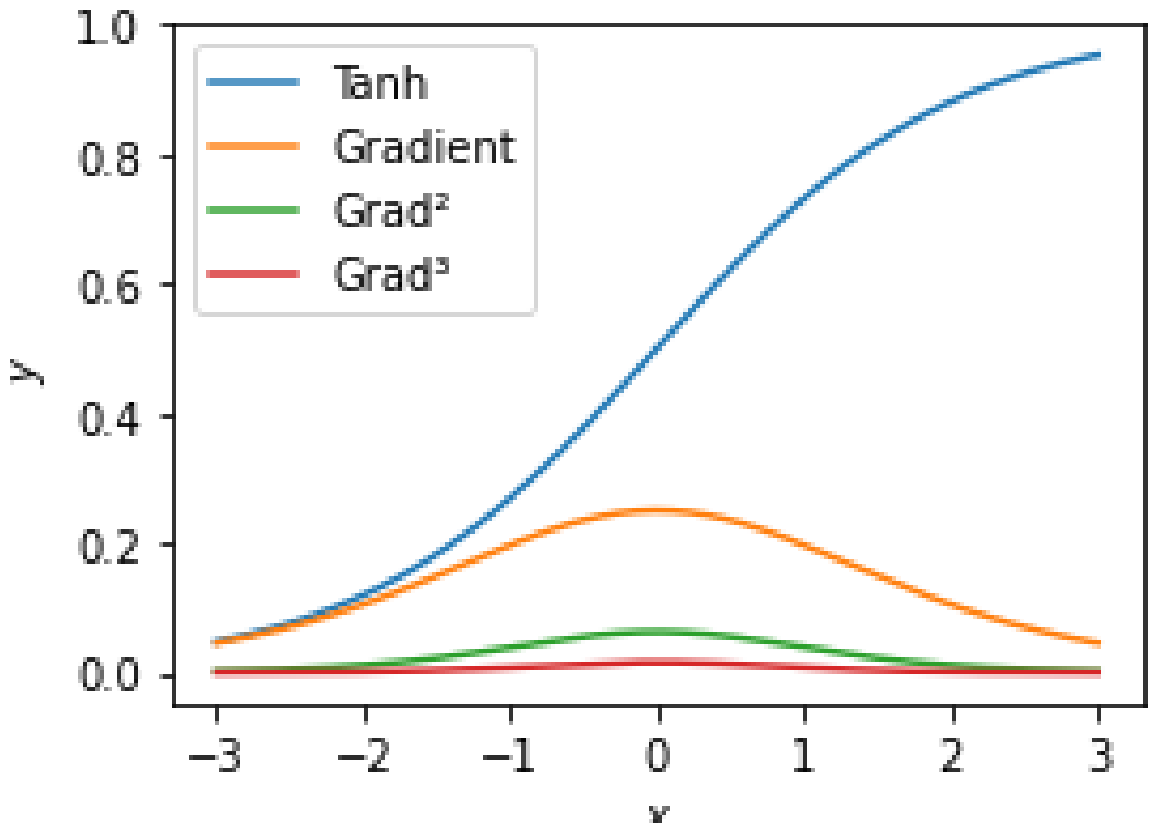
Glorot and Bengio 2010. Understanding the difficulty of training deep feedforward neural networks

El problema de gradiente explotando o desapareciendo



$$\frac{\partial J}{\partial w_{11}^{(0)}} = \frac{\partial J}{\partial x_1^{(3)}} \frac{\partial x_1^{(3)}}{\partial x_1^{(2)}} \frac{\partial x_1^{(2)}}{\partial x_1^{(1)}} \frac{\partial x_1^{(1)}}{\partial w_{11}^{(0)}}$$

Asumiendo que tenemos funciones de activación sigmoides (o tanh) en las capas internas y en el mejor de los casos cuando estan fasadas:



El problema de gradiente explotando o desapareciendo

ReLU, adaptative learning rate o gradientes conjugados, normalización batch son algunos de los aleviantes (pero el problema persiste).
Tanh and sigmoid cause huge vanishing gradient problems. En general no deberían ser usadas en deep networks.

Manteniendo la desviación estandard de la salida de las funciones de activación en 1 nos permite agregar mas capas en la red neuronal sin que los gradientes exploten o desaparezcan.
Es necesario inicializar los pesos a valores cercanos a 1.

Glorot, X. and Bengio, Y., 2010, March. Understanding the difficulty of training deep feedforward neural networks.

He, K., Zhang, X., Ren, S. and Sun, J., 2015. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification.

Inicialización de los pesos

Para eso debemos inicializar los parámetros (pesos) de la red con

- ReLU $\rightarrow$ inicialización: Kaiming-He
- Tanh $\rightarrow$ inicialización: Xavier-Glorot

Inicialización de Kaiming He

$$\mathbb{E}\left[\left(x_j^{(k+1)}\right)^2\right] = \sigma_w^2 \sum_{i=0}^{N_k} \mathbb{E}\left[\left(x_i^{(k)}\right)^2\right] = \sigma_w^2 \sum_{i=0}^{N_k} \frac{\sigma_x^2}{2} = \frac{1}{2} N_k \sigma_w^2 \sigma_x^2$$

Si quiero que $\mathbb{E}\left[\left(x_j^{(k+1)}\right)^2\right] = \sigma_x^2$ entonces $\sigma_w^2 = \frac{2}{N_K}$

Inicialización de Xavier Glorot

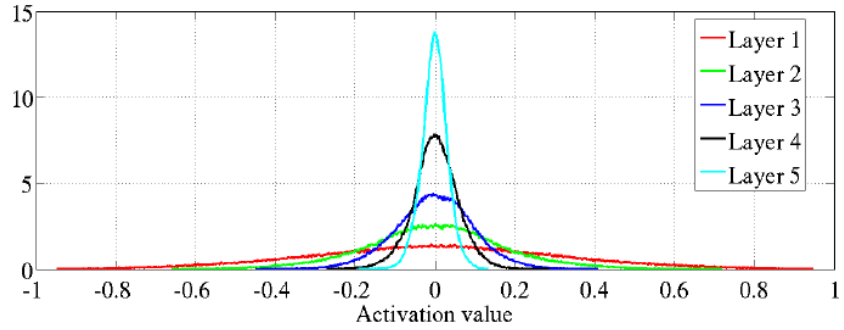
Pensando en sigmoid o tanh. Para una dada capa k obtiene dos condiciones:

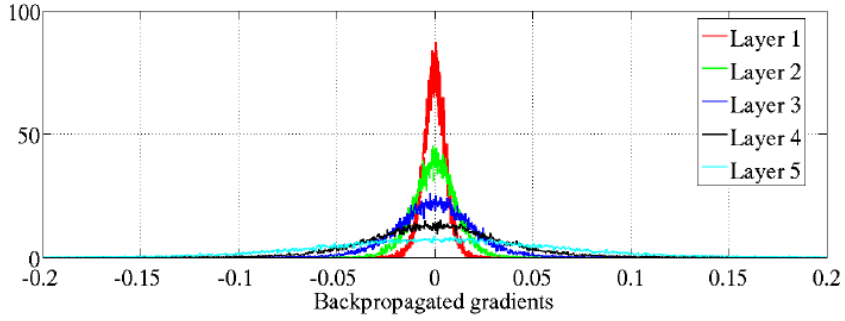
1. Condición Forward: $N_k \left(\sigma_w^{(k)}\right)^2 = 1$
2. Condición Backward: $N_{k+1} \left(\sigma_w^{(k)}\right)^2 = 1$

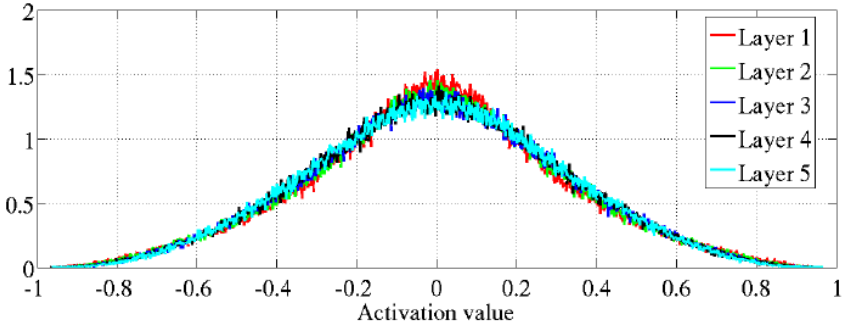
Inicialización de los pesos

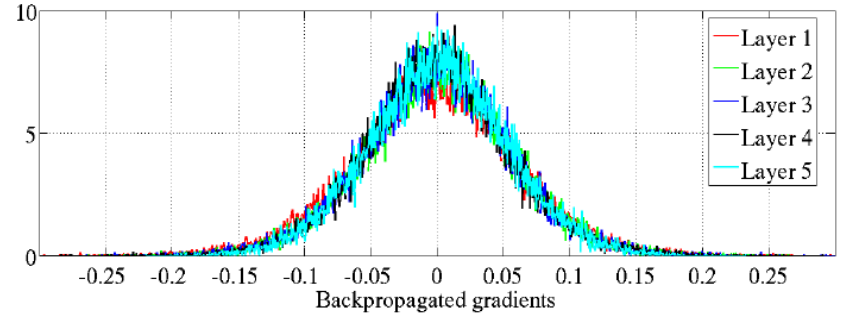
Para que satisfagan ambas condiciones:

$$w \sim U\left[-\frac{\sqrt{6}}{\sqrt{N_k + N_{k+1}}}, \frac{\sqrt{6}}{\sqrt{N_k + N_{k+1}}}\right]$$









Propagación hacia adelante. Propagación hacia atrás

Amplitud de las variables en distintas capas