UNIVERSIDAD NACIONAL
DEL NORDESTE

CONICET

I M I T

Modulo 7:

Aprendiendo de los datos con redes neuronales

¿Porque los modelos lineales son limitados?

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".

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.

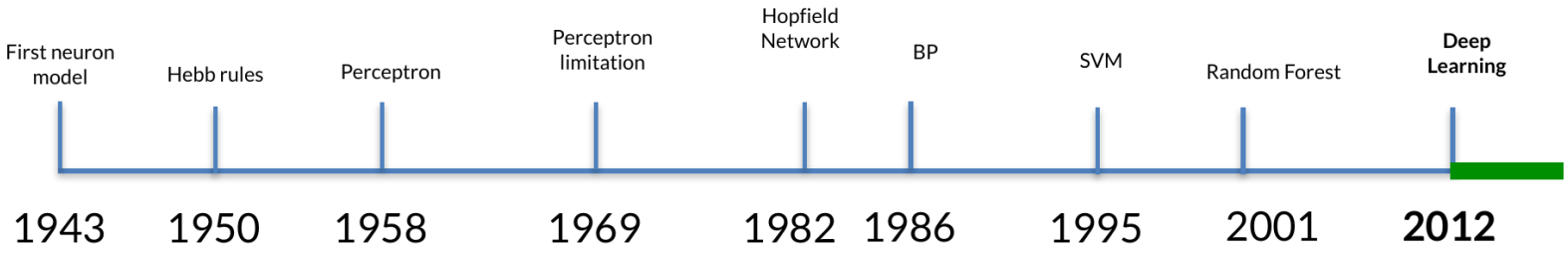
Objetivos

- Redes completamente conectadas (Fully connected)
- Rol de los nodos/neuronas y las funciones de activación
- Backpropagación
- Métodos de optimización
- Capacidad del modelo, over y underfitting
- Early stop y regularización
- Ejemplo general en pytorch

Bibliografía

- Goodfellow, I., Bengio, Y. and Courville, A., 2016. Deep learning. MIT press.
- Zhang, A., Lipton, Z.C., Li, M. and Smola, A.J., 2021. Dive into deep learning. arXiv preprint arXiv:2106.11342.
- Bishop, C.M. 1996. Pattern recognition and machine learning. Springer.

Historias de las redes neuronales



El hito de la revolución

MIT Technology Review

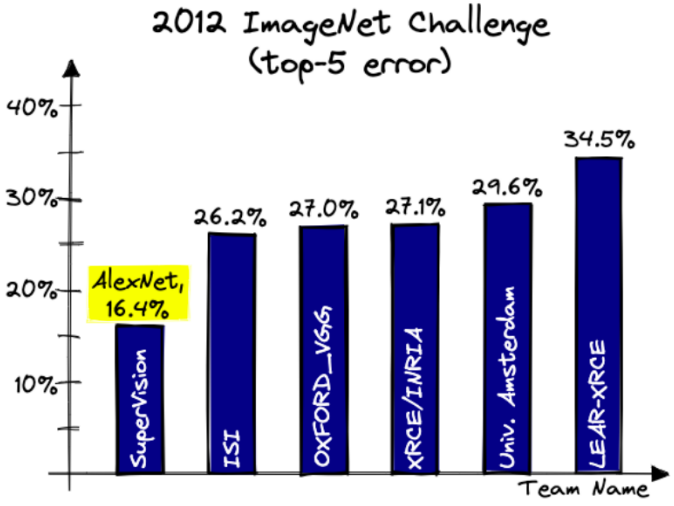
SUBSCRIBE

Best of 2014: The Revolutionary Technique That Quietly Changed Machine Vision Forever

In September, computer scientists revealed that machines are now almost as good as humans at object recognition; and the turning point occurred in 2012.

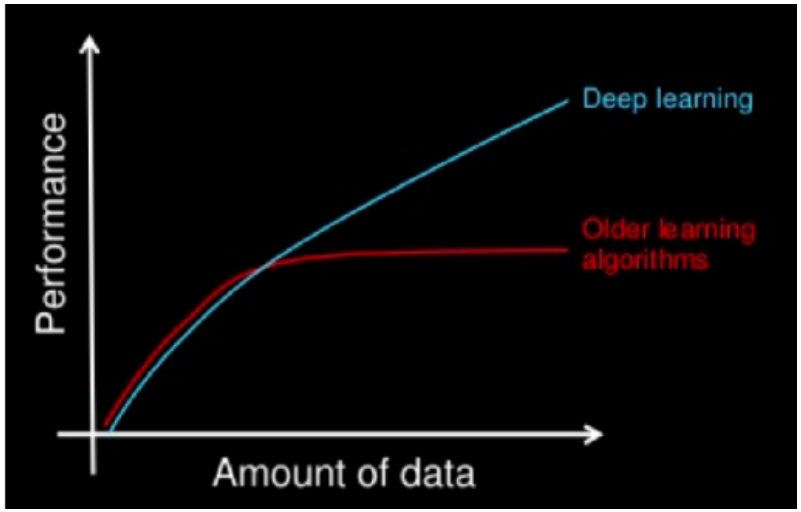
MIT news

Inicio de la revolución.

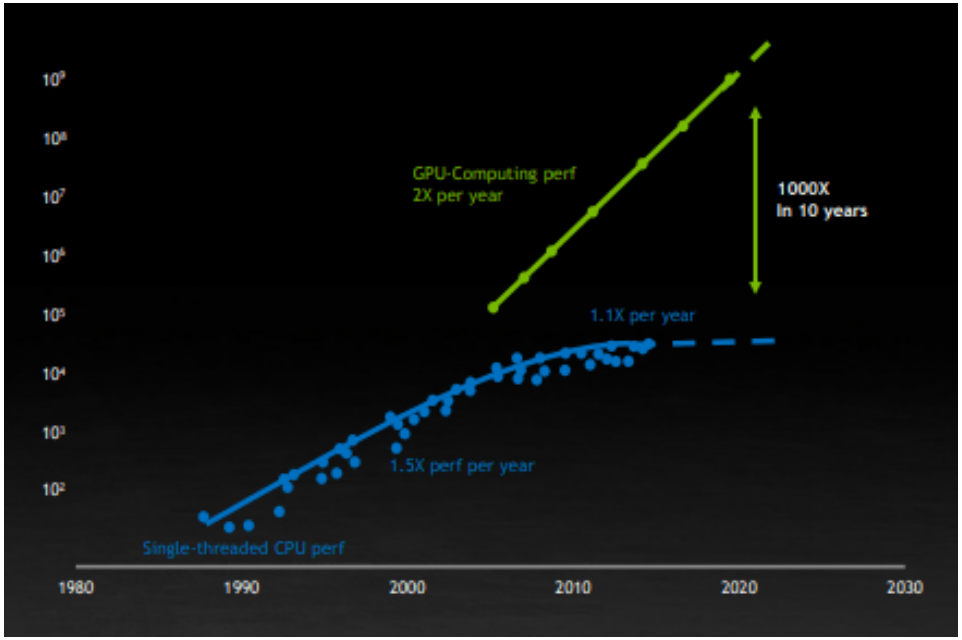


Krizhevsky A, Sutskever I, Hinton GE, 2012 Imagenet classification with deep convolutional neural networks. 120.000 Citas!!!

Causas de la revolución



Fuente Seeking alpha.

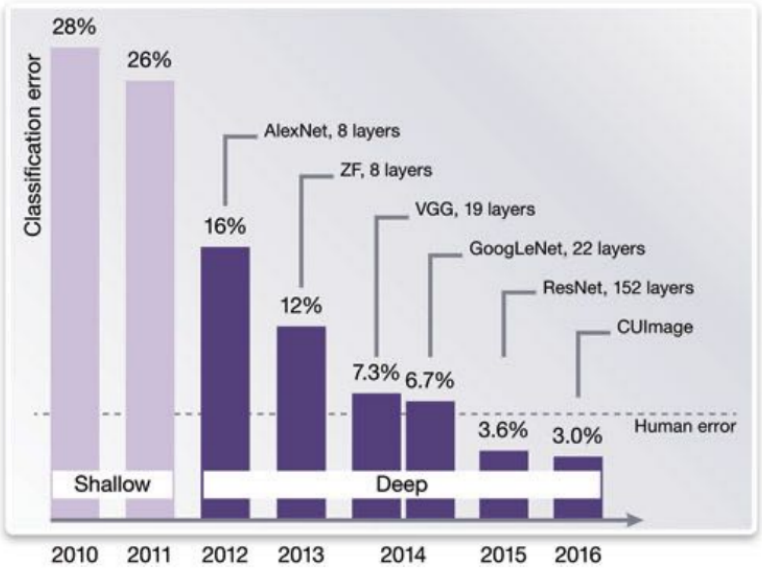


Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	Frontier - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, Oak Ridge National Laboratory, USA.	8,699,904	1,194.00	1,679.82	22,703
2	Supercomputer Fugaku - Supercomputer Fugaku, A64FX 48C 2.2GHz, Fujitsu RIKEN Center for Computational Science, Japan.	7,630,848	442.01	537.21	29,899
3	LUMI - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, HPE EuroHPC/CSC, Finland.	2,220,288	309.10	428.70	6,016
4	Leonardo - BullSequana XH2000, Xeon Platinum 8358 32C 2.6GHz, NVIDIA A100 SXM4 64 GB, Quad-rail NVIDIA HDR100 Infiniband, Atos EuroHPC/CINECA Italy.	1,824,768	238.70	304.47	7,404
5	Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, IBM DOE/SC/Oak Ridge National Laboratory, USA.	2,414,592	148.60	200.79	10,096
6	Sierra - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Mellanox DOE/NNSA/LLNL, USA.	1,572,480	94.64	125.71	7,438
7	Sunway TaihuLight – Sunway MPP, National Supercomputing Center in Wuxi, China.	10,649,600	93.01	125.44	15,371
8	Perlmutter - HPE Cray EX235n, AMD EPYC 7763 64C 2.45GHz, NVIDIA A100 SXM4 40 GB, Slingshot-10, HPE DOE/SC/LBNL/NERSC, USA.	761,856	70.87	93.75	2,589
9	Selene - NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Nvidia NVIDIA Corporation, USA.	555,520	63.46	79.22	2,646

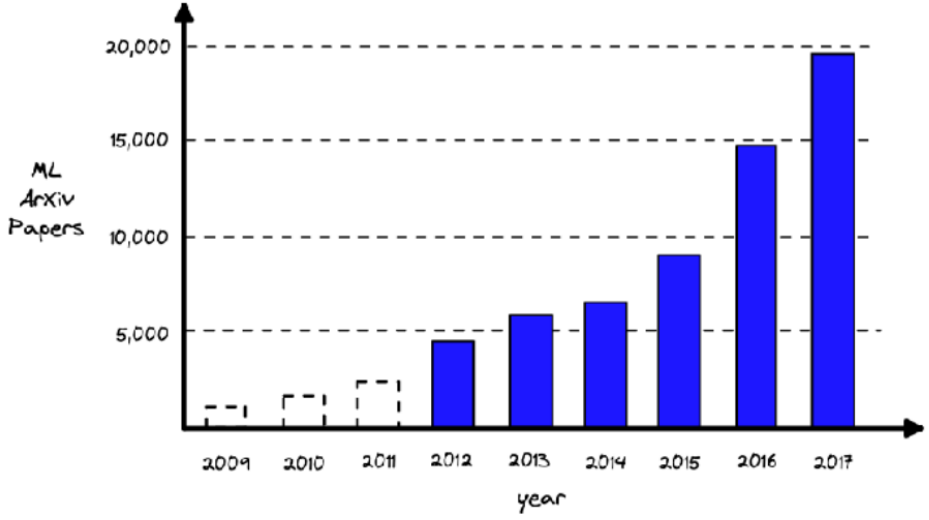
Top 500

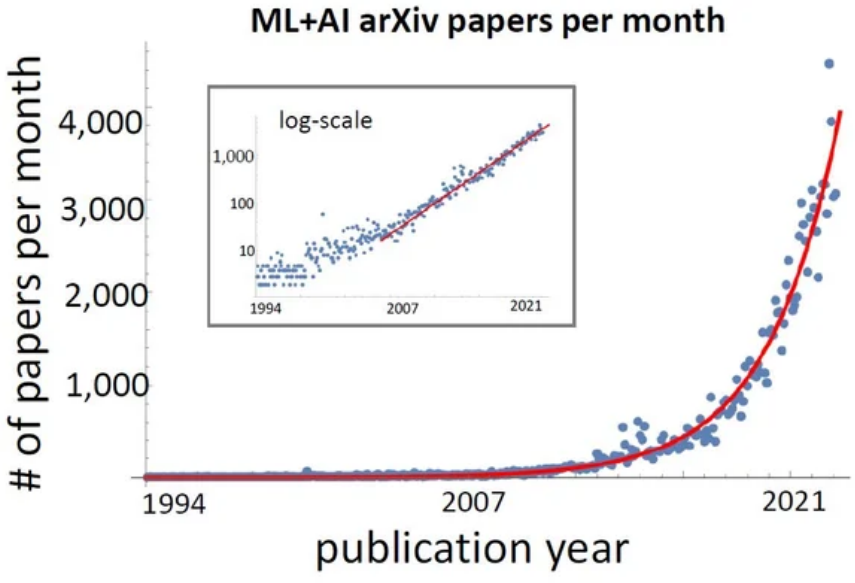
Fugaku

Evolución a partir de 2012

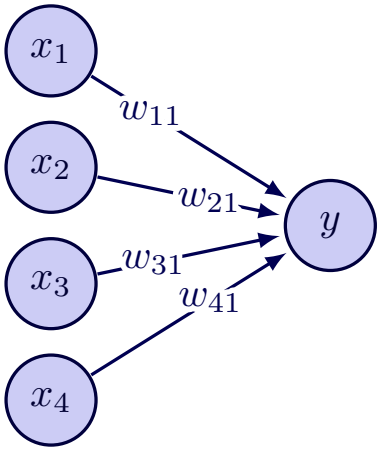


Fuente Reddit.





Red lineal. Multiples inputs - una salida.



- $x_1, \ldots, x_4$ son las entradas.
- $w_{11}, \ldots, w_{41}$ son los pesos
- y es la salida de la red.
- b es el sesgo o bias (ordenada al origen).

Cada variable de la red se denomina nodo (o neurona)

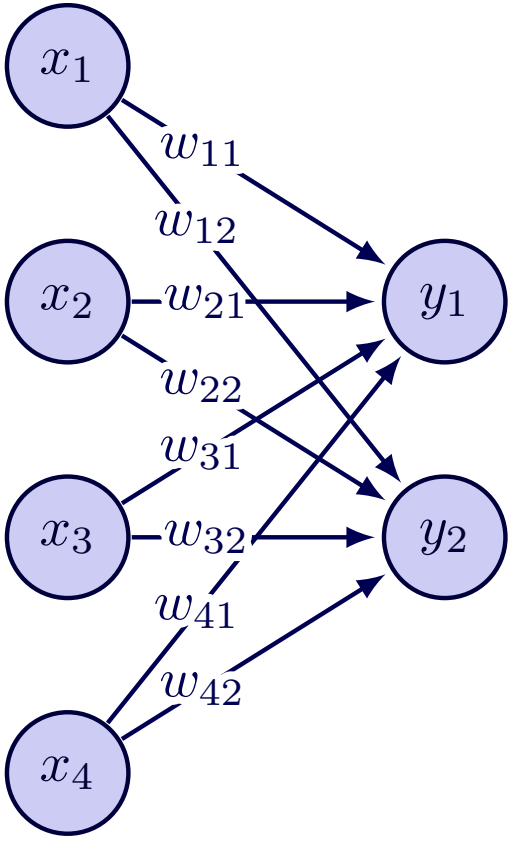
La operación que estoy realizando es un producto punto:

$$y = w_{11}x_1 + w_{21}x_2 + w_{31}x_3 + w_{41}x_4 + b = \mathbf{w} \cdot \mathbf{x} + b$$

Puedo pensar en una variable de entrada mas, fija en $x_0 = 1$, de esta manera $b = w_{01}$,

$$y = \mathbf{w} \cdot \mathbf{x}$$

Red lineal. Multiples inputs - multiples salidas.



- 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: $\mathbf{y} = \mathbf{W}\,\mathbf{x}$

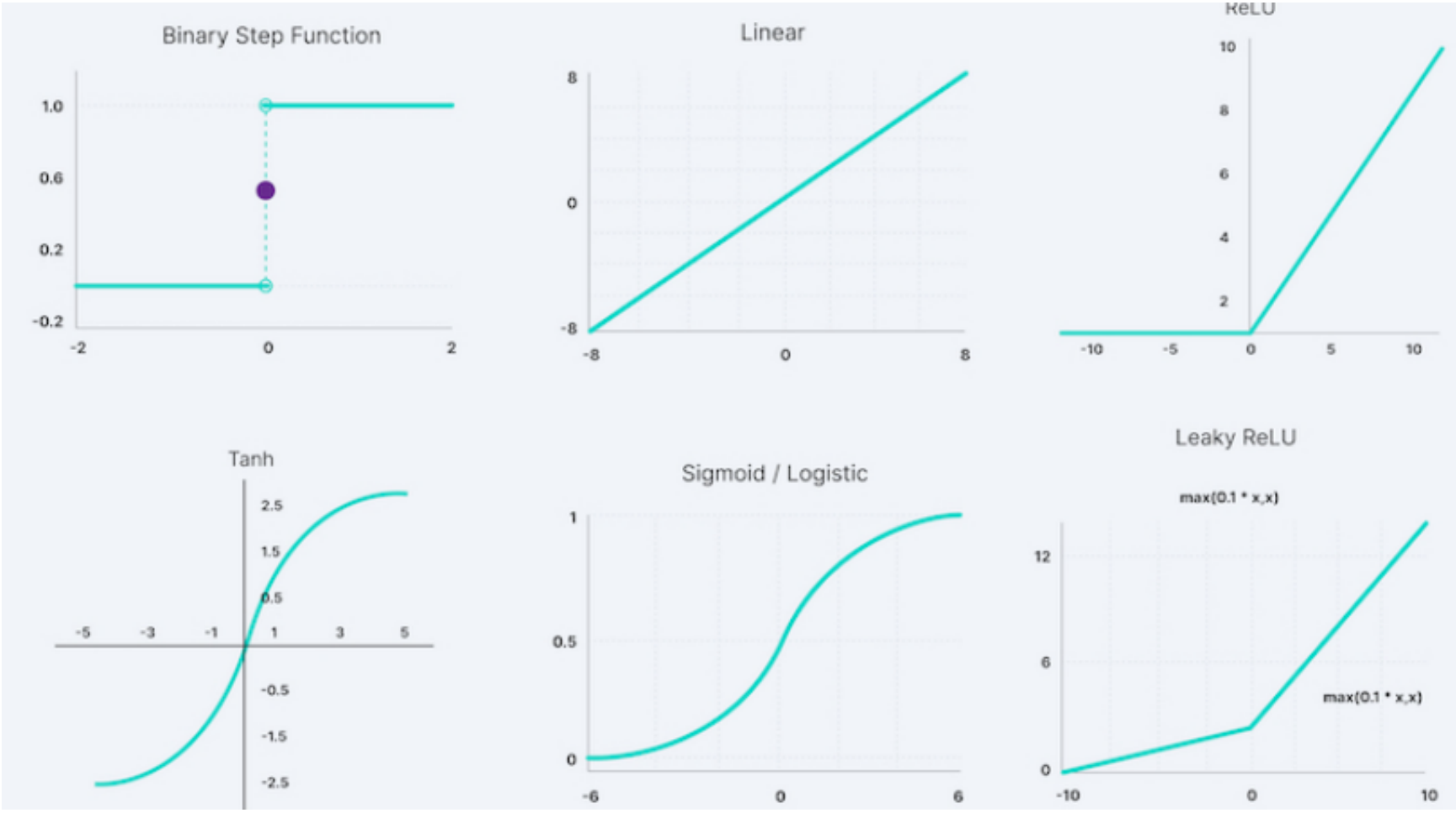
Agregando no-linealidad

Todo lo que hacemos es agregar un función no-lineal h a las redes lineales ya definidas:

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

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.
Se denomina función de activación.

Funciones de activación



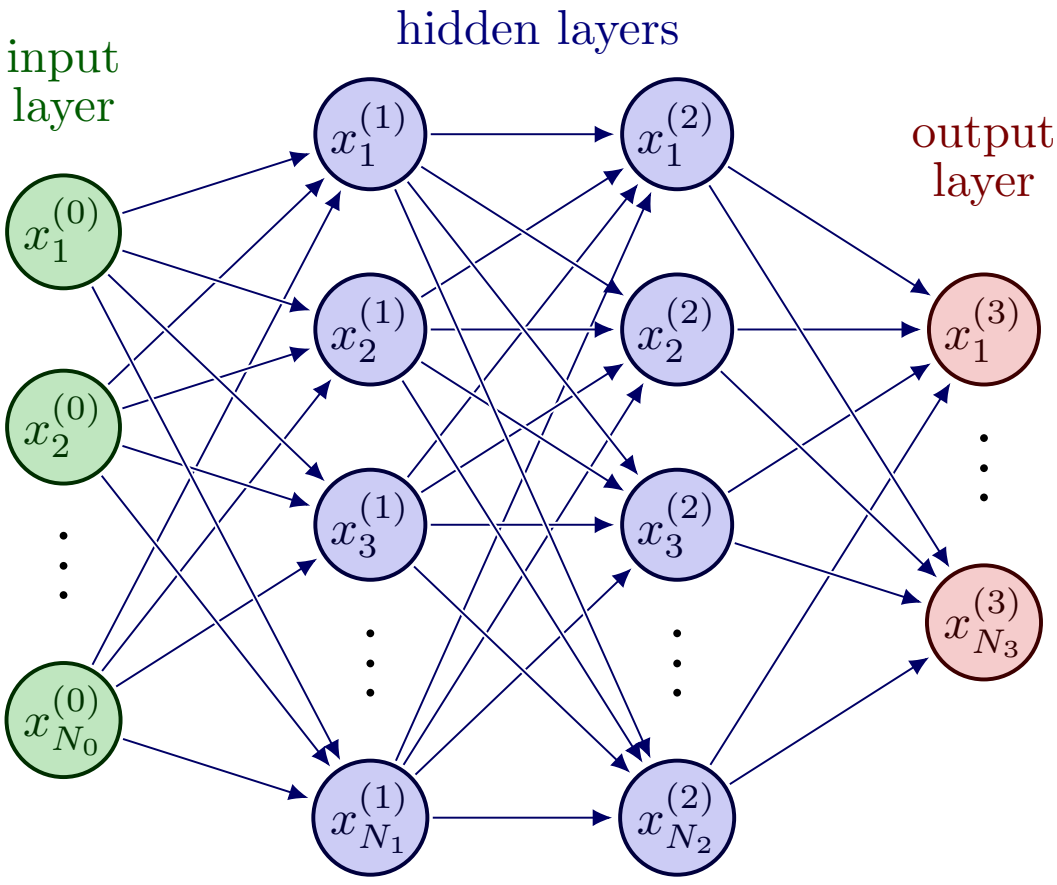
Magia de las redes de las redes neuronales

Composición de funciones.

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

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

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:

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

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.

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:

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

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

Propagación hacia adelante

Capa k-esima:

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

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:

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

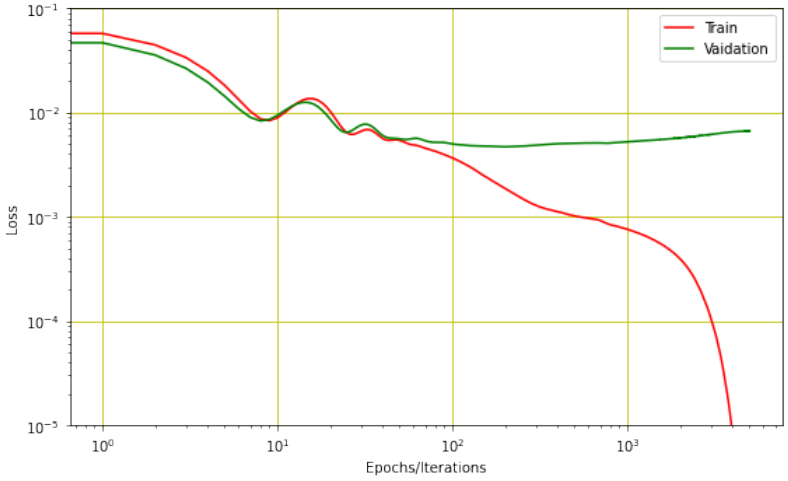
Input $\to$ Capa 2 $\to$ Capa 3 $\to$ Output $\to$ Función de pérdida $J(\text{Input}, \text{Output})$

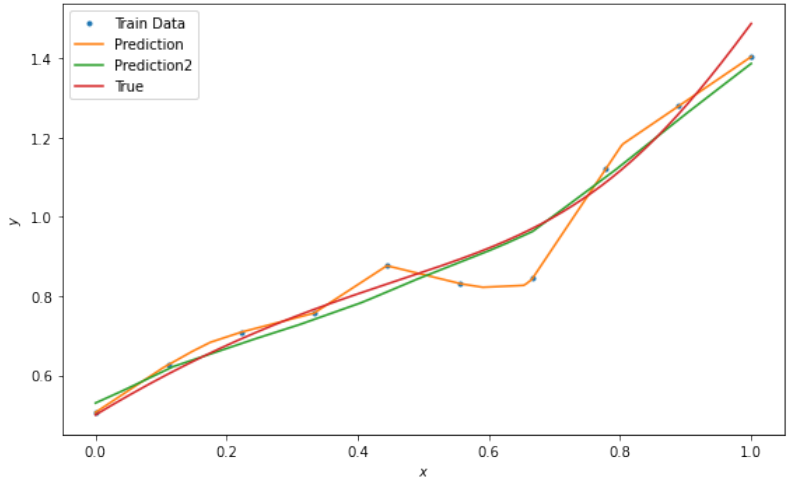
$$\begin{aligned} J(\mathbf{W}) &= \frac{1}{2N_d}\sum_{n=1}^{N_d}\left\|\mathbf{x}_n^{(K)} - \mathbf{y}_n\right\|^2 \\ &= \frac{1}{2N_d}\sum_{n=1}^{N_d}\left\|h^{(K-1)}\left(\mathbf{W}^{(K-1)} h^{(K-2)}\left(\cdots h^{(0)}\left(\mathbf{W}^{(0)}\mathbf{x}_n^{(0)}\right)\cdots\right)\right) - \mathbf{y}_n\right\|^2 \end{aligned}$$

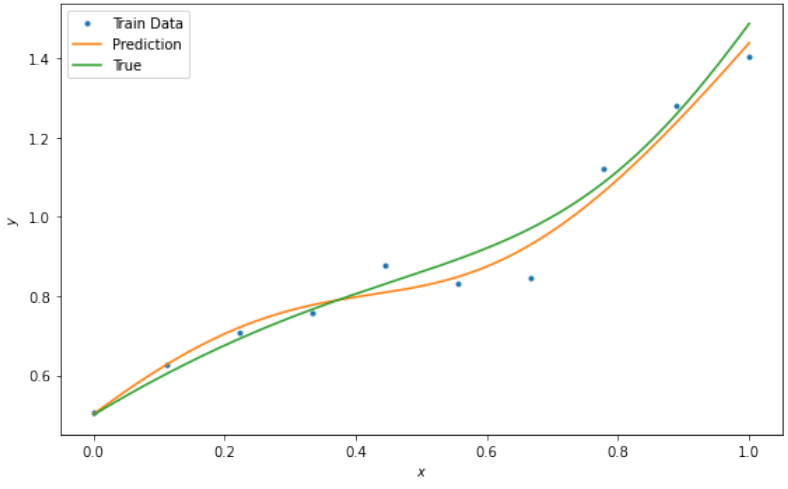
Tamaño del espacio de control FC $\mathbf{W}$: $\sum_{k=0}^{K-1} N_w^{(k)} = \sum_{k=0}^{K-1} N_k N_{k+1}$

Optimización con una red neuronal

Red 1-100-1







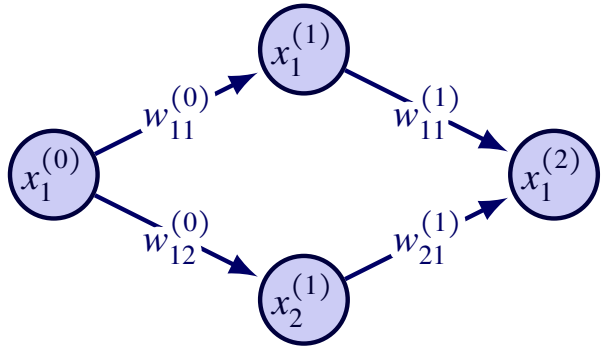
Fn Activación:

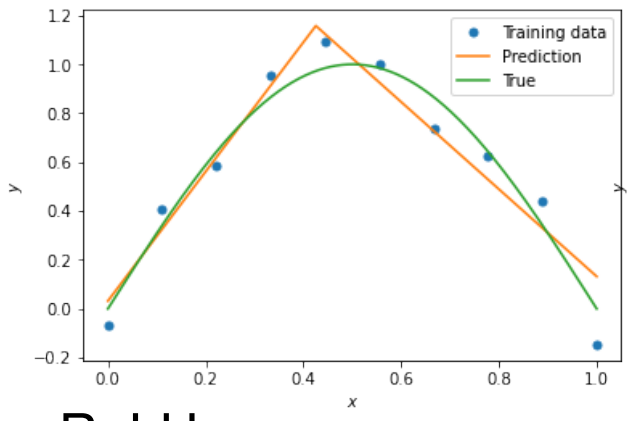
- Arriba: ReLU
- Abajo: Tanh

Funciones de activación

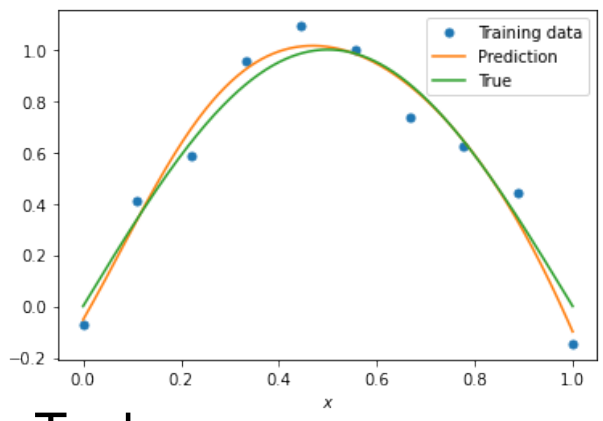
Red ultra sencilla 1-2-1 (2 neuronas internas).

Funciones de activación en ambos enlaces.

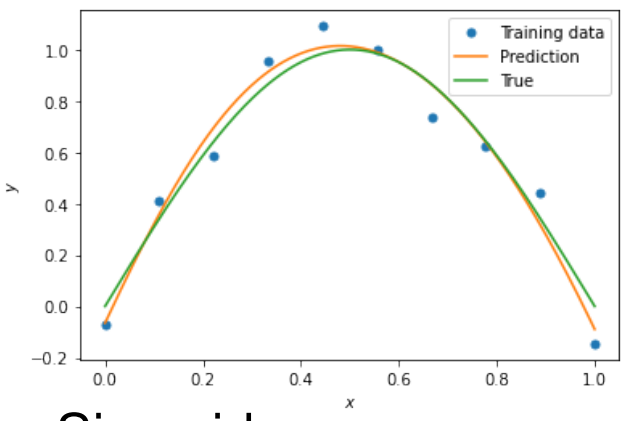




ReLU



Tanh



Sigmoid

¿Que puede decir de los parámetros w?

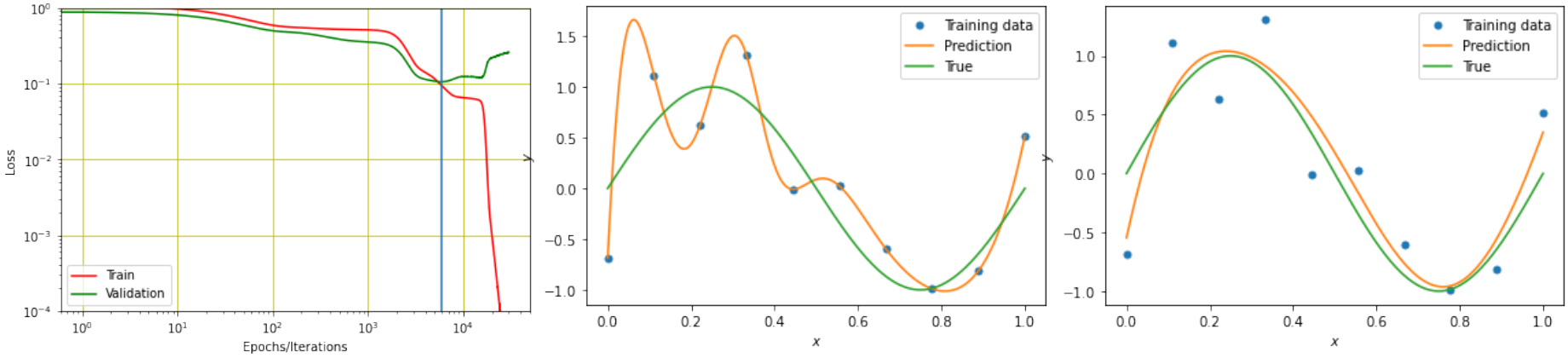
¿Que roles tienen las funciones de activación en cada capa?

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.

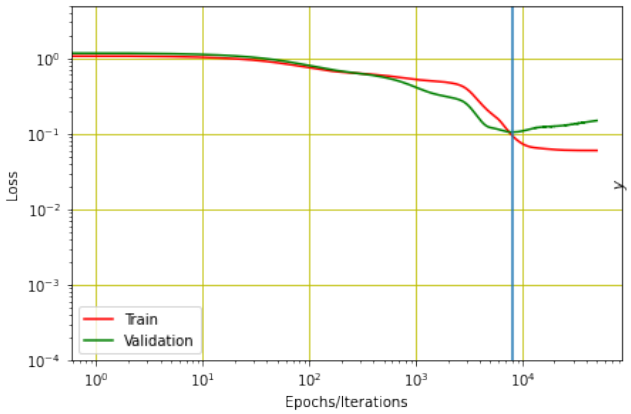


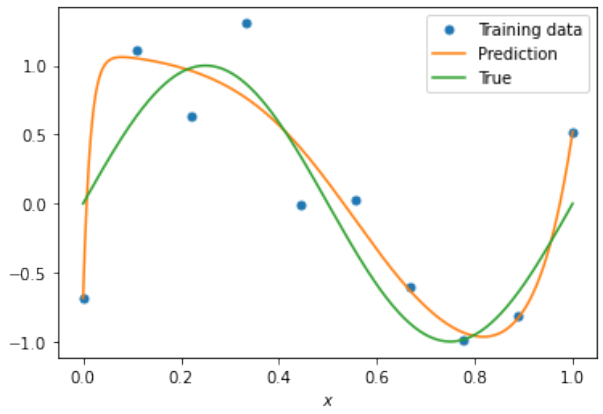
Demasiada complejidad del modelo.

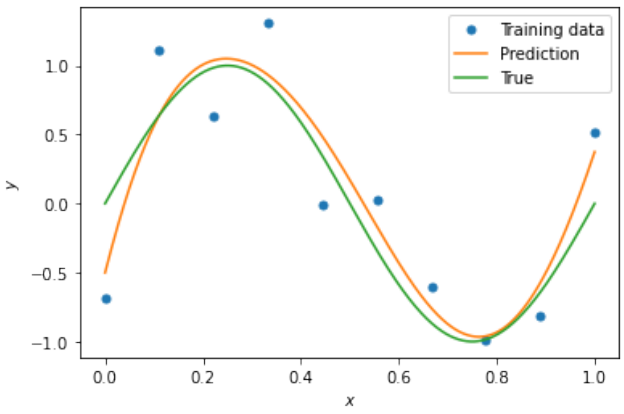
- Fuerte ajuste de los datos de entrenamiento (Izq) pero sin poder de generalización
- Detengo el entrenamiento cuando la J de validación empieza a crecer (Der).

¿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).

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, $q = 1$.

Esta implementado en pytorch (argumento de la función de optimización).
Weight decay: Se cambian directamente los parametros en el paso de update (no la J).

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

Determinación del gradiente.

Todos los algoritmos de optimización requieren el gradiente.

¿Como determinamos el gradiente en una red neuronal?

Gradiente de la funcion de pérdida.

Vectorialmente $\mathbf{x}_0 \in \mathbb{R}^{N_0}$, $\mathbf{x}_1 \in \mathbb{R}^{N_1}$: $\mathbf{y} \in \mathbb{R}^{N_1}$

$$J(\mathbf{w}, \mathbf{x}_0) = \frac{1}{2}\|\mathbf{x}_1 - \mathbf{y}\|^2; \qquad \mathbf{x}_1 = f(\mathbf{w}, \mathbf{x}_0)$$

Derivo con respecto a $\mathbf{x}_1$,

$$\delta J = (\mathbf{x}_1 - \mathbf{y})\delta\mathbf{x}_1$$

Con respecto a la variable de entrada y los parámetros

$$\delta\mathbf{x}_1 = \frac{\partial \mathbf{x}_1}{\partial \mathbf{x}_0}\delta\mathbf{x}_0 + \frac{\partial \mathbf{x}_1}{\partial \mathbf{w}}\delta\mathbf{w}$$

$$\frac{\partial J}{\partial \mathbf{w}} = \frac{\partial J}{\partial \mathbf{x}_1}\frac{\partial \mathbf{x}_1}{\partial \mathbf{w}}$$

Se requiere determinar como se propagan las perturbaciones desde la J hasta los pesos de la red. Con $K+1$ capas:

$$\frac{\partial J}{\partial \mathbf{w}} = \frac{\partial J}{\partial \mathbf{x}_K}\frac{\partial \mathbf{x}_K}{\partial \mathbf{x}_{K-1}}^{\top} \cdots \frac{\partial \mathbf{x}_2}{\partial \mathbf{x}_1}^{\top}\frac{\partial \mathbf{x}_1}{\partial \mathbf{w}}$$

Variables con las perturbaciones - variables adjuntas.

Ej. Modelo cuadrático.

$$J(x_0) = \frac{1}{2}(x_1 - y_1)^2, \quad x_1 = \alpha x_0^2$$

Si queremos ver el impacto de una perturbación de x_0 en J

$$\delta x_1 = x_0^2 \delta\alpha, \quad \delta x_1 = 2\alpha x_0 \delta x_0, \quad \delta J = (x_1 - y_1)\delta x_1$$

Podemos pensar en variables que nos propagan los errores/perturbaciones:

$$\tilde{x}_1 = \tilde{x}_1 + (x_1 - y_1)$$

$$\tilde{x}_0 = \tilde{x}_0 + 2\alpha x_0 \tilde{x}_1$$

Con respecto a α:

$$\tilde{\alpha} = \tilde{\alpha} + x_0^2 \tilde{x}_1$$

Gradiente de la función de pérdida de una red

Learning representations by back-propagating errors

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

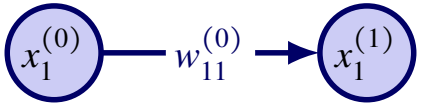
* Institute for Cognitive Science, C-015, University of California, San Diego, La Jolla, California 92093, USA
† Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Philadelphia 15213, USA

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.

Propagación hacia adelante. Tangente lineal.

Comencemos por una conexión simple:



$$x_1^{(1)} = f(x_1^{(0)}; w_{11}^{(0)}) = h\left(w_{11}^{(0)} x_1^{(0)}\right)$$

¿Cómo se propaga una perturbación del parámetro en esta conexión?

$$\delta x_1^{(1)} = \frac{\partial f(x_1^{(0)}; w_{11}^{(0)})}{\partial w_{11}^{(0)}} \delta w_{11}^{(0)} = \frac{\partial h(z)}{\partial z} x_1^{(0)} \delta w_{11}^{(0)}$$

En el caso del input:

$$\delta x_1^{(1)} = \frac{\partial f(x_1^{(0)}; x_1^{(0)})}{\partial x_1^{(0)}} \delta x_1^{(0)} = \frac{\partial h(z)}{\partial z} w_{11}^{(0)} \delta x_1^{(0)}$$

Estas perturbaciones impactan sobre la función de pérdida:

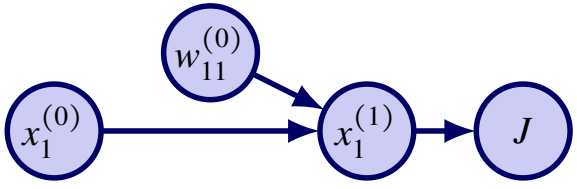
$$\delta J = (x_1^{(1)} - y) \frac{\partial h}{\partial z} \left(w_{11}^{(0)} \delta x_1^{(0)} + x_1^{(0)} \delta w_{11}^{(0)}\right)$$

Aplicación intensiva de la regla de la cadena. Memoria vs costo computacional.

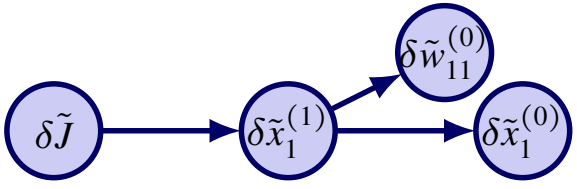
Backpropagation. Conexión simple

Tenemos que desandar la propagación hacia adelante. Que una perturbación producida en la J (output) nos diga cuanto fue causado por la perturbación de un parámetro (input).

Forward



Backward



$$\delta\tilde{x}_1^{(1)} = \frac{\partial J}{\partial x_1^{(1)}} + \delta\tilde{J}$$

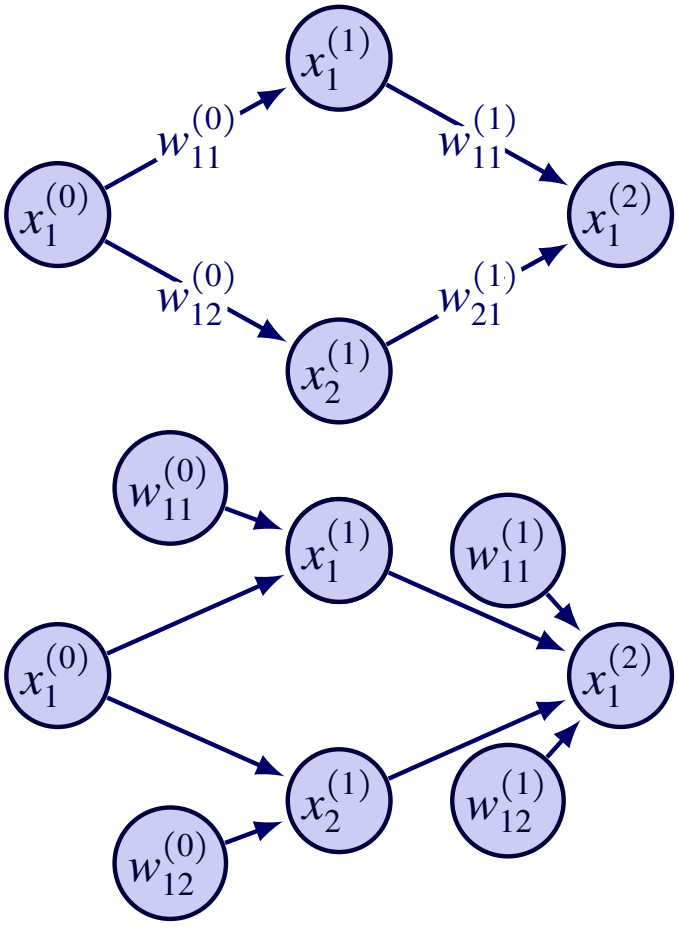
$$= (x_1^{(1)} - y) + \delta\tilde{J}$$

$$\delta\tilde{w}_{11}^{(0)} = \frac{\partial x_1^{(1)}}{\partial w_{11}^{(0)}}\delta\tilde{x}_1^{(1)} = \frac{\partial h}{\partial z}\, x_1^{(0)}\, \delta\tilde{x}_1^{(1)}$$

$$\delta\tilde{x}_1^{(0)} = \frac{\partial x_1^{(1)}}{\partial x_1^{(0)}}\delta\tilde{x}_1^{(1)} = \frac{\partial h}{\partial z}\, w_{11}^{(0)}\, \delta\tilde{x}_1^{(1)}$$

Interpretación: El TL nos mezcla todas las perturbaciones para conducir al δJ. El Backprop me produce las contribuciones de cada parámetro.

Backprop en una bifurcación



$$\begin{bmatrix} \delta\tilde{x}_1^{(1)} \\ \delta\tilde{x}_2^{(1)} \end{bmatrix} = \begin{bmatrix} \frac{\partial x_1^{(2)}}{\partial x_1^{(1)}} \\ \frac{\partial x_1^{(2)}}{\partial x_2^{(1)}} \end{bmatrix} \delta\tilde{x}_1^{(2)}$$

$$\delta\tilde{x}_1^{(0)} = \begin{bmatrix} \frac{\partial x_1^{(1)}}{\partial x_1^{(0)}} & \frac{\partial x_1^{(1)}}{\partial x_2^{(0)}} \end{bmatrix} \begin{bmatrix} \delta\tilde{x}_1^{(1)} \\ \delta\tilde{x}_2^{(1)} \end{bmatrix}$$

Los pesos no propagan información hacia atras (no requieren rastreos).

Si nos interesa $w_{11}^{(0)}$ lo obtenemos por:

$$\delta\tilde{w}_{11}^{(0)} = \frac{\partial x_1^{(1)}}{\partial w_{11}^{(0)}} \delta\tilde{x}_1^{(1)}$$

En redes profundas por las nolinealidades para evaluar $\frac{\partial x_i^{(k)}}{\partial x_j^{(k-1)}}$ requerimos los valores de los nodos internos: x_i^k

Backpropagation: Gradientes en pytorch y tensorflow

Modulo: autograd

Torch nos calcula el backprop en forma automatica!

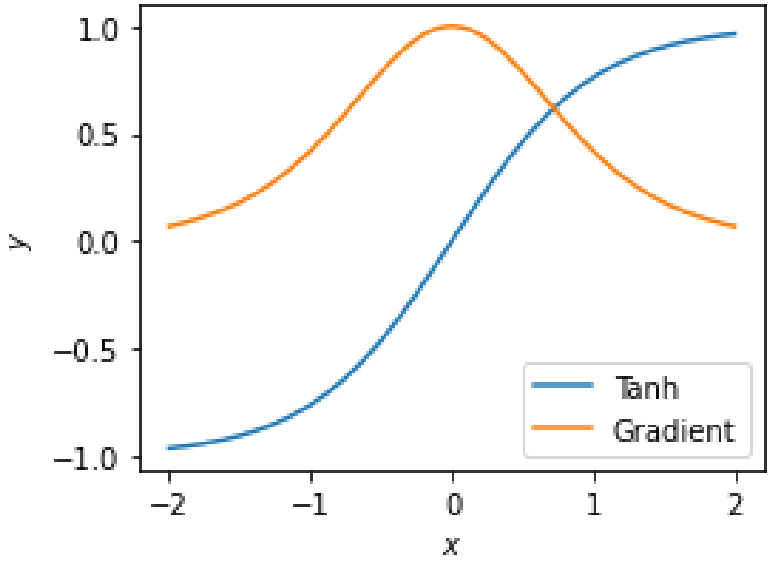
Para que autograd realice la diferenciación automática se debe especificar:

```
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 hasta la variable/parámetro independiente.

Analogía con asimilación de datos 4DVar

En la asimilación de datos variacional también se define una función objetivo por ejemplo:

$$\begin{aligned} J(\mathbf{x}_k^0) &= J^b(\mathbf{x}_k^0) + J^o(\mathbf{x}_k^0) \\ &= \frac{1}{2}\left(\mathbf{x}_k^0 - \mathcal{M}(\mathbf{x}_{k-1}^a)\right)^\top \mathbf{B}^{-1}\left(\mathbf{x}_k^0 - \mathcal{M}(\mathbf{x}_{k-1}^a)\right) + \\ &\qquad \frac{1}{2}\left(\mathbf{y}_{k+1} - \mathcal{H}(\mathcal{M}(\mathbf{x}_k^0))\right)^\top \mathbf{R}^{-1}\left(\mathbf{y}_{k+1} - \mathcal{H}(\mathcal{M}(\mathbf{x}_k^0))\right) \end{aligned}$$

Modelo forward = modelo dinámico: Evoluciona tiempos:

$$t_{k-1} \rightarrow t_k, \qquad \mathbf{x}_k = \mathcal{M}(\mathbf{x}_{k-1})$$

Se requiere minimizar J donde $\mathbf{x}_0$ es el estado. $\mathbf{x} \in 10^8 - 10^9 dim$.
Para calcular el gradiente de la función objetivo/de costo (como parte de la minimización):

$$\nabla_{\mathbf{x}_k^0} J$$

Modelo adjunto. Integración del adjunto

Para calcular el gradiente de J,

$$\nabla_{\mathbf{x}_0} J_b(\mathbf{x}_0) = (\mathbf{P}^f)^{-1}(\mathbf{x}_0 - \mathbf{x}_0^f)$$

Con respecto a la función de costo de las observaciones

$$\mathbf{d}_k = \mathbf{R}^{-1}\left(\mathbf{y}_k - \mathcal{H}(\mathbf{x}_k)\right); \qquad \mathbf{x}_k = \sum_{i=0}^{k} \mathcal{M}(\mathbf{x}_i)$$

$$\nabla_{\mathbf{x}_k} J = 2\mathbf{H}_k^\top \mathbf{d}_k$$

Comenzamos desde último tiempo k y luego evolucionamos hacia atrás hasta $i = 0$

$$\nabla_{\mathbf{x}_0} J = \sum_{k=0}^{N} \mathbf{M}_0^\top \cdots \mathbf{M}_{k-1}^\top (2\mathbf{H}_k^\top \mathbf{d}_k)$$

$$\mathbf{M}_k = \left.\frac{\partial \mathcal{M}}{\partial \mathbf{x}}\right|_{x_k}$$

Expandiendo la suma

$$\nabla J_o = \mathbf{H}_0^\top \mathbf{d}_0 + \mathbf{M}_1^\top [\mathbf{H}_1^\top \mathbf{d}_1 + \mathbf{M}_2^\top [\mathbf{H}_2^\top \mathbf{d}_2 + \cdots + \mathbf{M}_N^\top 2\mathbf{H}_N^\top d_N]]$$

Autograd $\rightarrow$ TAMC (Tangent and Adjoint Model compiler)?

Optimización

Matriz Hessiana

La aproximación cuadrática de un función multivariada alrededor de $\hat{\mathbf{w}}$ viene dada por

$$J(\mathbf{w}) \approx J(\hat{\mathbf{w}}) + (\mathbf{w} - \hat{\mathbf{w}})^\top \nabla J\big|_{\hat{\mathbf{w}}} + \frac{1}{2}(\mathbf{w} - \hat{\mathbf{w}})^\top \mathbf{H} (\mathbf{w} - \hat{\mathbf{w}})$$

donde definimos la matriz Hessiana,

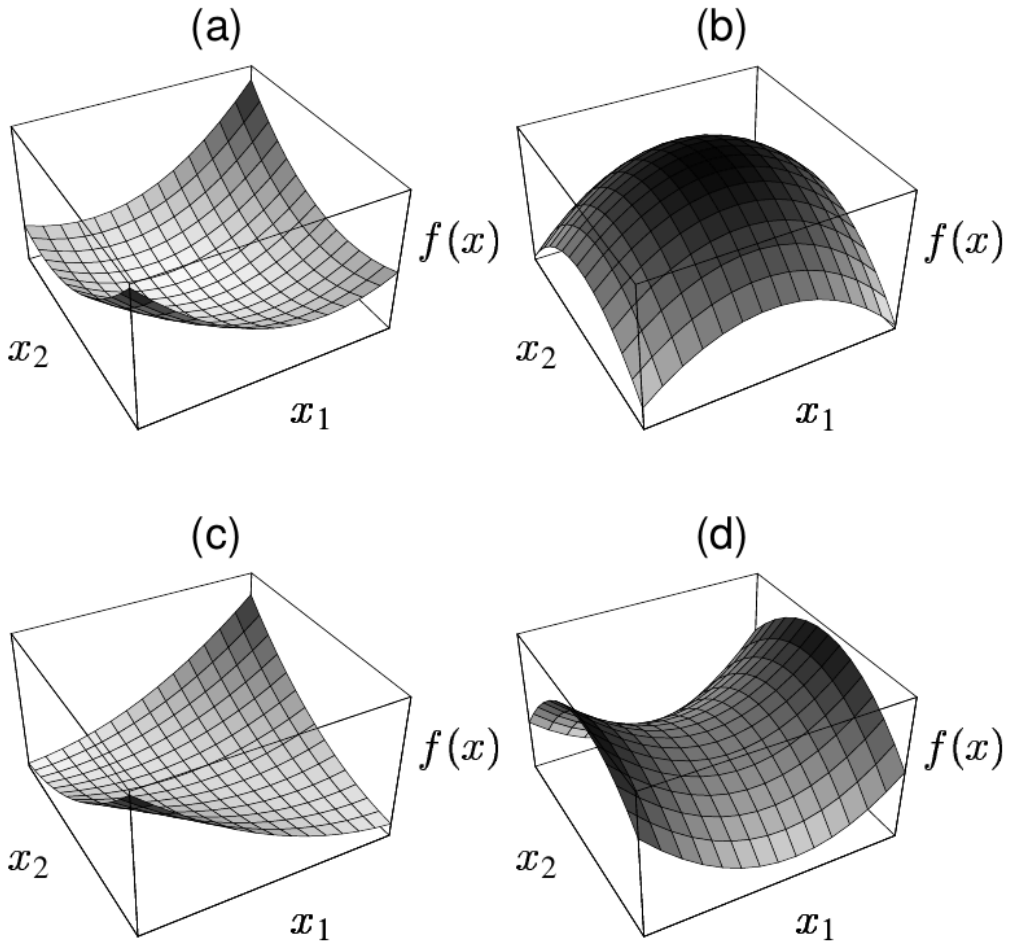
$$(\mathbf{H})_{ij} \doteq \left. \frac{\partial J}{\partial w_i w_j} \right|_{\hat{\mathbf{w}}}$$

En funciones multivariadas el análisis de los puntos críticos $\nabla J(\mathbf{w}^*) = 0$ se realiza a través del Hessiano:

Si $\mathbf{H}(\mathbf{w}^*)$ es definido positivo, entonces $\mathbf{w}^*$ es un mínimo.

$\mathbf{H}$ es definido positivo si todos los autovalores son positivos.

Caracterización 2d de la matriz Hessiana



Función de forma cuadrática en 2d con distintas Hessianas.

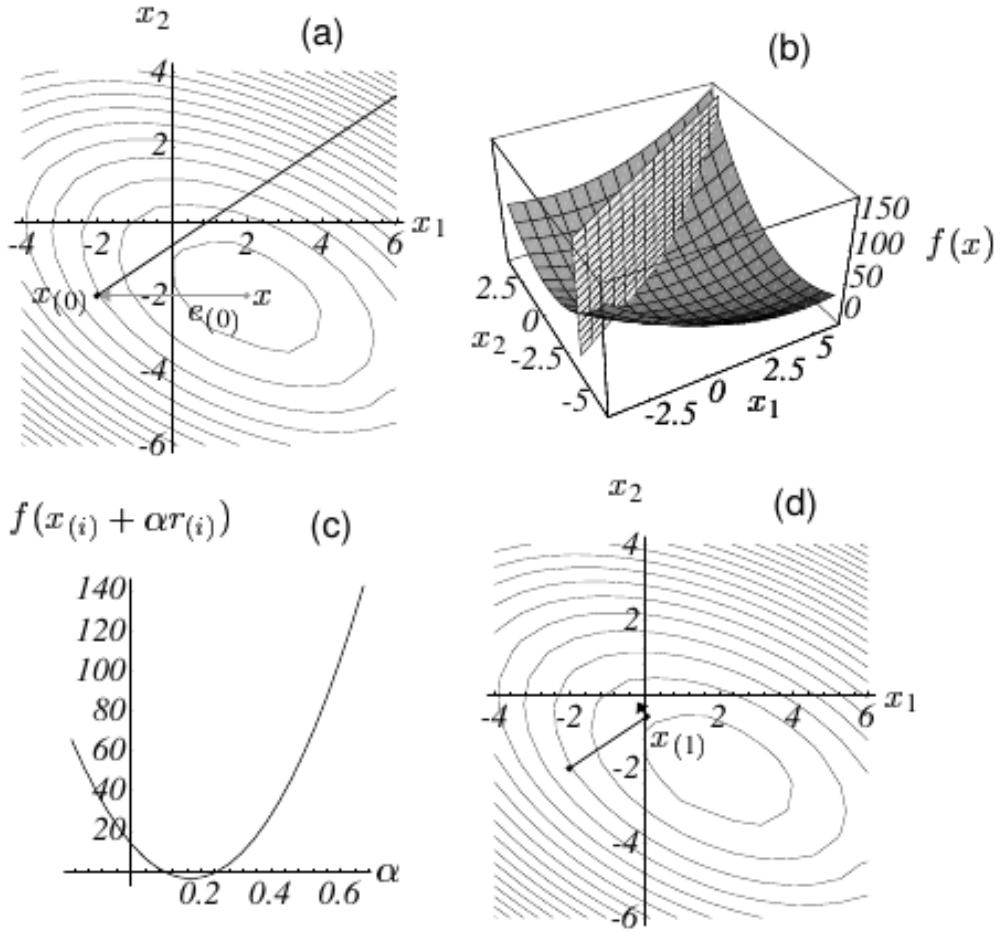
a) Matriz hessiana definida positiva.

b) Matriz definida negativa.

c) Matriz singular positiva indefinida (línea de infinitas soluciones).

d) Matriz indefinida con un punto de ensilladura.

Optimización por descenso de gradientes



Comenzamos desde [-2,-2] y hacemos un paso en la dirección de maximo descenso.

Descenso de gradientes con todos los datos

Considerando que $J = \frac{1}{2N_D}(\mathbf{Y} - \mathbf{X}\mathbf{w})^\top(\mathbf{Y} - \mathbf{X}\mathbf{w})$ el gradiente debe ser realizado con toda la base de datos:

$$\mathbf{w} = \hat{\mathbf{w}} - \eta \nabla_{\mathbf{w}} J(\mathbf{X}, \mathbf{Y})\Big|_{\hat{\mathbf{w}}}$$

```
for i in range ( n_epochs ):
    dJ_dw = gradient ( loss_function , data , w )
    w = w - learning_rate * dJ_dw
```

Para cada evaluación del gradiente tenemos que considerar a todos los datos.

No requerimos de evaluaciones de la función de costo (para la optimización!).

Optimización por descenso de gradientes en pytorch

```
# Considerando datos X_train, y_train
model = torch.nn.Linear(n_inputs, n_outputs) # Linear Function/Net
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4) # Instancio tecnica de
                                                optimizaci\'on
J = torch.nn.MSELoss() # Defino funci\'on de costo

for i in range(n_epochs):

    y_pred = model(X_train)    # Calculate prediction from x

    loss = J(y_pred,y_train)  # Calculate loss
    optimizer.zero_grad() # Initialize gradients
    loss.backward()        # Backprop
    optimizer.step()      # Update parameters
```

Estamos evaluando la base de datos completa en cada época

Descenso de gradientes estocásticos

$$\mathbf{w} = \hat{\mathbf{w}} - \eta \nabla_{\mathbf{w}} J\big|_{\hat{\mathbf{w}}} (\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$$

SGD: Lo que hacemos es tomar una muestra de todo el conjunto de datos en cada iteración.

```
for i in range ( n_epochs ):
    np.random.shuffle ( data )
    for xy in data:
        dJ_dw = gradientJ ( loss_function , xy , w )
        w = w - learning_rate * dJ_dw
```

Fundamentación

- El descenso de gradientes estocásticos converge casi seguramente a un mínimo global (para funciones convexas) o a un mínimo local en general.
- Para garantizar la convergencia la learning rate debe ser pequeña.
- Esto es una consecuencia del teorema de Robbins-Siegmund (1971).
- Online forms of the EM algorithm: Neal and Hinton (1996) incremental EM algorithm.

Descenso de gradientes por mini-batch

$$\mathbf{w} = \hat{\mathbf{w}} - \eta \nabla_{\mathbf{w}} J\big|_{\hat{\mathbf{w}}} (\mathbf{x}^{(i:i+n)}, \mathbf{y}^{(i:i+n)})$$

donde n es el tamaño del mini-batch.

```
for i in range ( n_epochs ):
    np.random.shuffle ( data )
    for batch in get_Batches(data, batch_size=64):
        dJ_dw = gradientJ ( loss_function , batch , w )
        w = w - learning_rate * dJ_dw
```

Época. Iteraciones. Evaluaciones de la función de costo.

Sigue siendo estocástico pero no tanto! Cuales son los pros y cons de esto?

Suffling

Es habitual dividir el conjunto de datos en: Entrenamiento (70%), Validación (15%), Testing (15%).

- El conjunto de datos es subdividido en mini-batches para todos los datos.
- En cada época recorremos todos los mini-batches cubriendo toda la base de datos
- Conviene mezclar los datos para que los mini-batches se conformen con distintos datos.
- shuffling: Mezcla aleatoria de los datos en cada epoca (los mini-batches van a tener distintos datos).

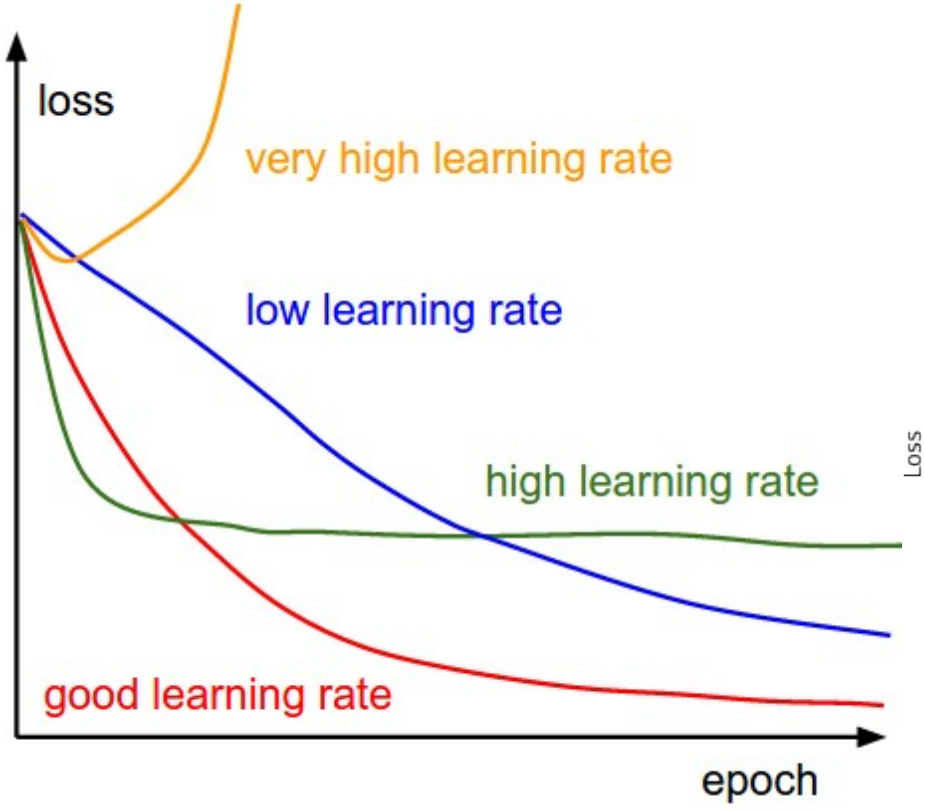
Suffling y mini-batches en pytorch

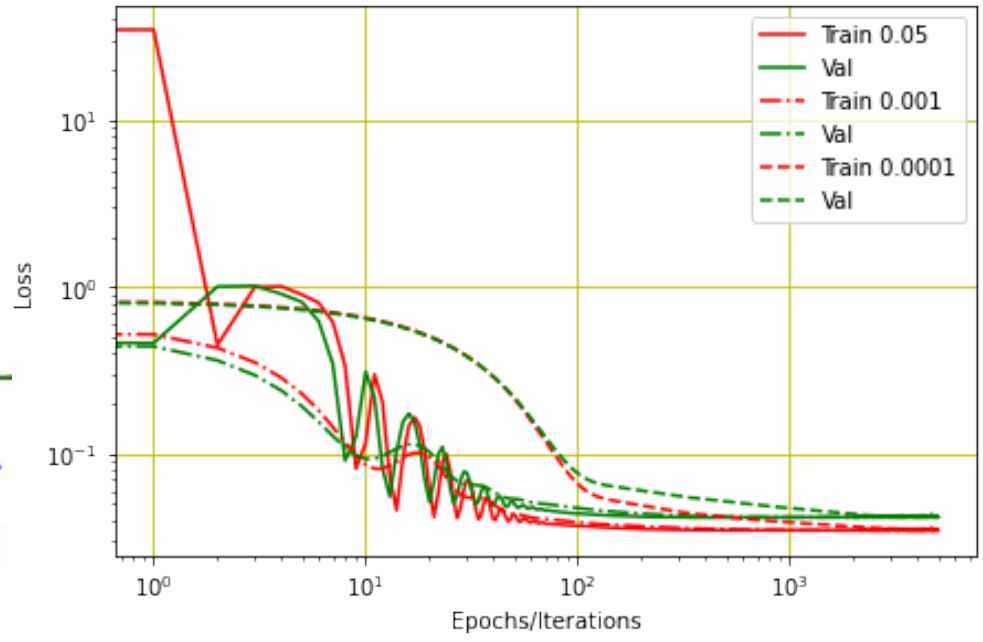
```
train_dat = torch.utils.data.TensorDataset(X, Y)
train_loader = torch.utils.data.DataLoader(train_dat, batch_size=64,
                                           shuffle=True, drop_last=True)
for iepoch in range(n_epochs):
    for x_batch, y_batch in train_loader:  #batches of the dataset

        y_f = NNmdl(x_batch) # prediction of the batch
        loss = Loss(x_batch, y_batch, y_f ) # evaluate cost with the batch

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

Efecto de la tasa de aprendizaje

```
w = w - learning_rate *dJ_dw
```

Rango usual: `learning_rate=1.-3 /5.e-5`





Newton-Raphson

¿Como elijo la tasa de aprendizaje óptima?

El método de Newton 1D para encontrar raíces, por Taylor,

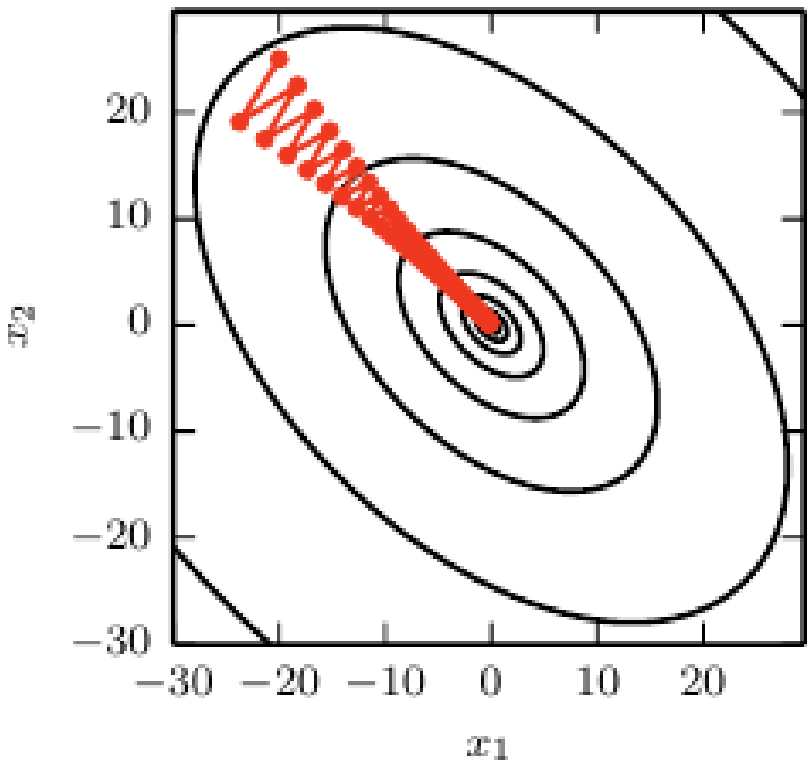
$$f(x_k) = 0 \approx f(x_{k-1}) + (x_k - x_{k-1}) f'(x_{k-1})$$

$$x_k = x_{k-1} - \frac{f(x_{k-1})}{f'(x_{k-1})}$$

La extensión a puntos críticos de una función J multivariada,

$$\mathbf{x}_k = \mathbf{x}_{k-1} - \mathbf{H}^{-1} \nabla J(x_{k-1})$$

Momento



- El SGD es particularmente ineficiente para funciones de pérdida con número de condición alto.
- Elipses/soides alargadas.
- Debemos aumentar la razón de aprendizaje en las direcciones con mayor descenso, basado en las altas derivadas.

$$v_t = \gamma v_{t-1} + \eta \nabla_{\mathbf{w}} J(\mathbf{w})$$

$$\mathbf{w}_t = \mathbf{w}_{t-1} - v_t$$

Adagrad

Dependiendo los parámetros y sus derivadas parciales nos gustaría realizar correcciones selectivas/adaptativas para cada uno.
Duchi et al 2011 proponen:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot \nabla_{\mathbf{w}} J(\mathbf{w}_t)$$

donde G_t es la suma de los cuadrados de las derivadas pasadas con respecto al parámetro correspondiente.
$\epsilon = 1.e - 8.$

RMSprop / Adadelta

Hacemos decaer exponencialmente la suma de derivadas cuadradas en cada dirección

$$K_t = \gamma K_{t-1} + (1-\gamma)\nabla_{\mathbf{w}} J(\mathbf{w}_t)^2$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \frac{\eta}{\sqrt{K_t + \epsilon}} \odot \nabla_{\mathbf{w}} J(\mathbf{w}_t)$$

```
def RMSprop(gradJ):
    ''' Adadelta learning.
         Hyperpar. gamma=0.9, eta0=1.e-2, epsilon=1.e-6  '''

    self.sum_gradsq=gamma * self.sum_gradsq + (1-gamma) * gradJ**2
    eta1=eta0/(epsilon+np.sqrt(self.sum_gradsq))

    return -eta1*gradJ # parameter update
```

Este esquema fue sugerido por Hinton. Con una normalización se obtiene el Adadelta.

Adam

Kingma and Ba (2014) proponen trabajar con ambos: los gradientes y los gradientes cuadrados (primer y segundo momento)

$$\mathbf{v}_t = \beta_1 \mathbf{v}_{t-1} + (1-\beta_1)\nabla_{\mathbf{w}} J(\mathbf{w}_t)$$

$$\mathbf{K}_t = \beta_2 \mathbf{K}_{t-1} + (1-\beta_2)[\nabla_{\mathbf{w}} J(\mathbf{w}_t)]^2$$

Se corrije el sesgo

$$\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1-(\beta_1)^t}$$

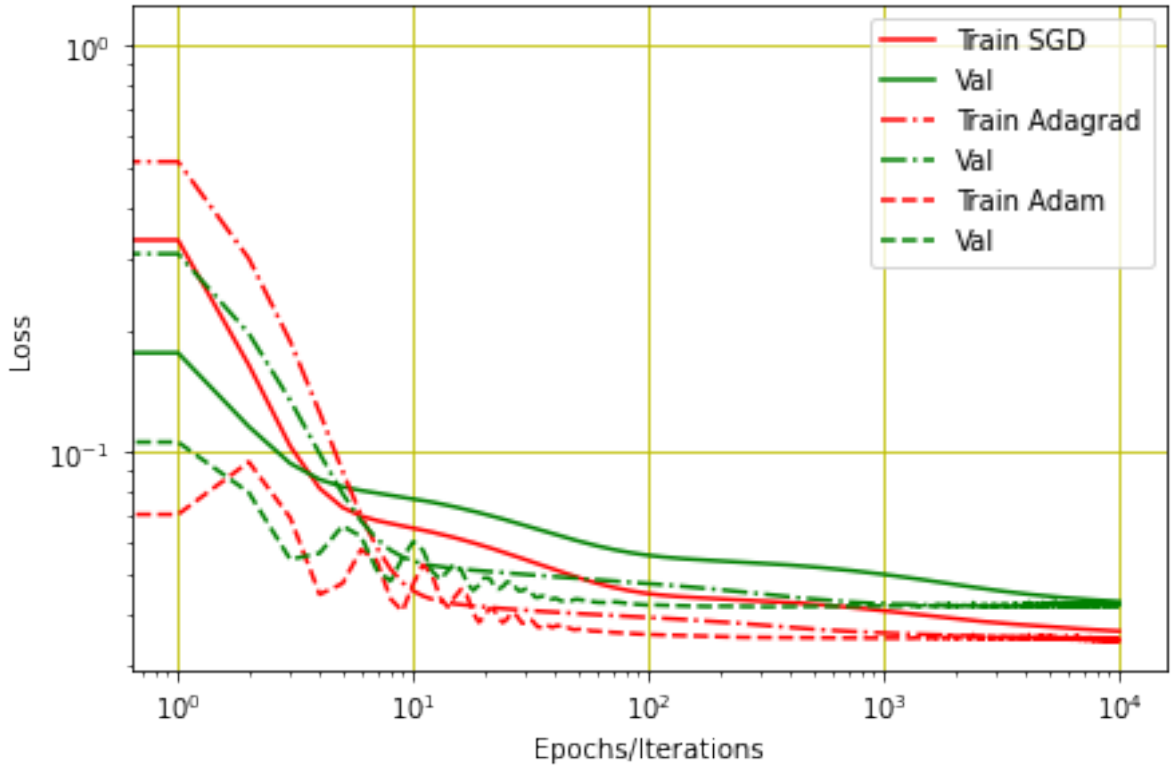
$$\hat{\mathbf{K}}_t = \frac{\mathbf{K}_t}{1-(\beta_2)^t}$$

Actualización de los parámetros

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \frac{\eta}{\sqrt{\hat{K}_t} + \epsilon} \odot \hat{\mathbf{v}}_t$$

Para datos reales limitados en comparación a los parámetros: AdamW Loshchilov, I. and Hutter, F., 2017. Decoupled weight decay regularization. arXiv preprint arXiv:1711.05101.

Comparación



Discusión:

https://www.ruder.io/optimizing-gradient-descent/

Precioso visualizador de los metodos:

https://github.com/lilipads/gradient_descent_viz

Métodos de (Quasi) segundo orden

- Conjugate gradients
- Métodos Quasi-Newton: L-BFGS
- Preconditioning

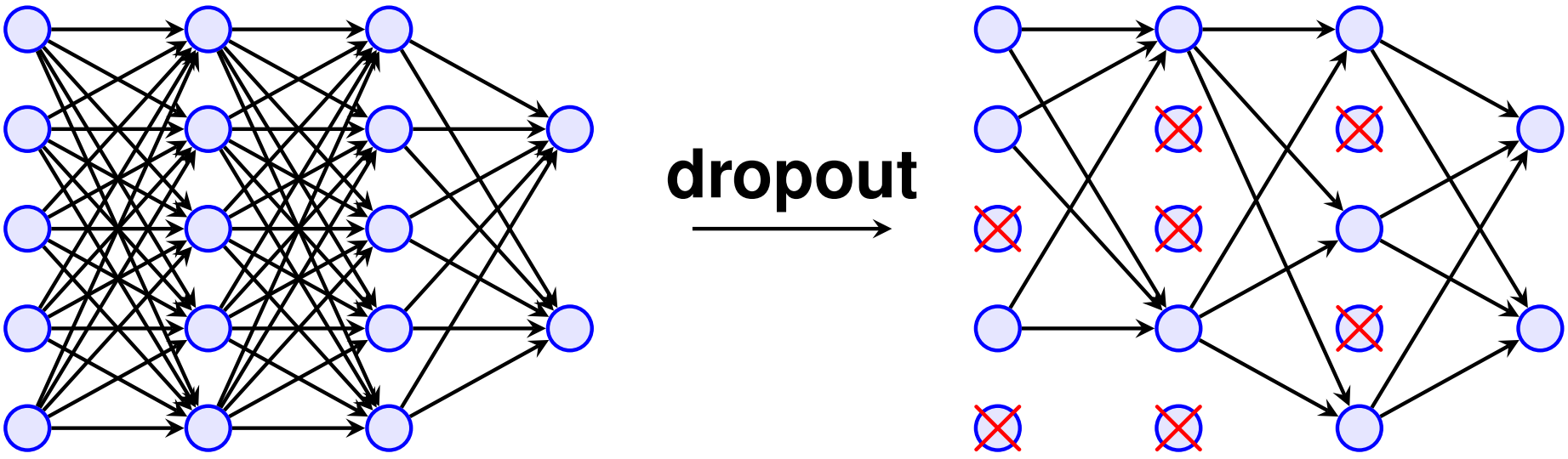
En pytorch L-BFGS se encuentra implementado. Con mini-batch y grandes dimensiones es complicada su aplicación en ML.

En problemas de asimilación de datos variacional, éstos son los métodos mas utilizados. Porque se conoce el Hessiano.

Dropout

Desaparición aleatoria de nodos de la red durante el aprendizaje.

Es una forma de regularización.



Gal, Y. and Ghahramani, Z., 2016. Dropout as a Bayesian approximation: Representing model uncertainty in deep learning. ICML.

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.

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

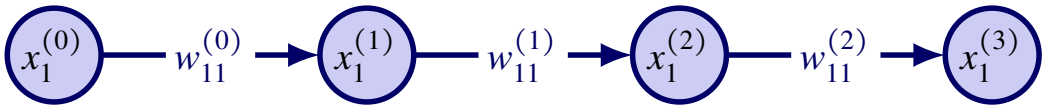
Si el producto de las derivadas del peso y de la función de activación en un nodo exceden el valor de uno estos van a continuar creciendo exponencialmente a lo largo del camino.

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. Hence, they should not be used in deep networks.

- keeping the standard deviation of layers activations around 1 will allow us to stack several more layers in a deep neural network without gradients exploding or vanishing.
- ReLU $\rightarrow$ inicialización: kaiming

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: