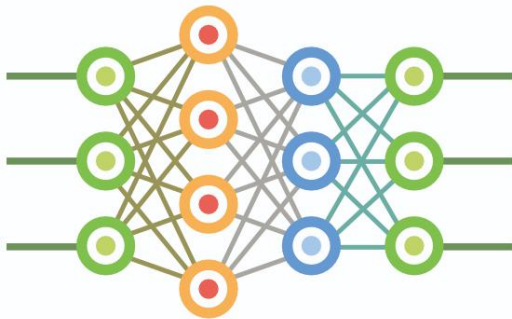
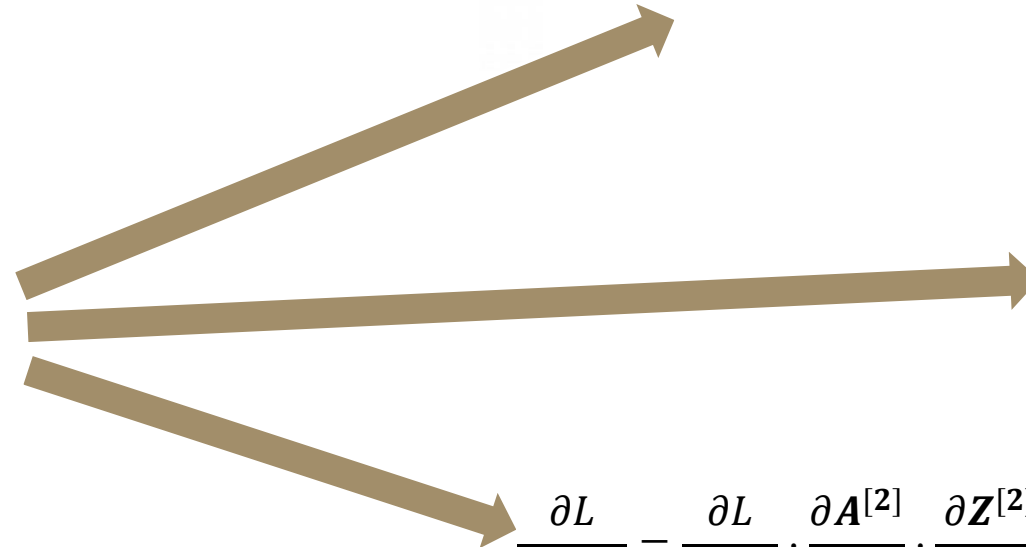


# AGENDA

 PyTorch

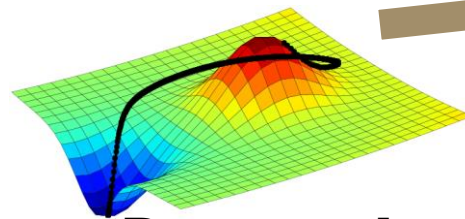


**Redes Neuronales  
Artificial (ANN)**

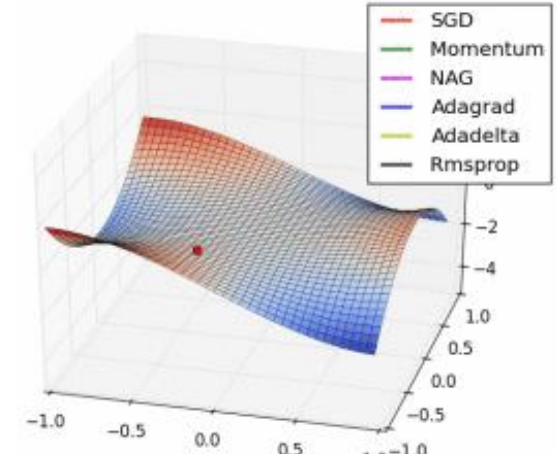


$$\frac{\partial L}{\partial W^{[2]}} = \frac{\partial L}{\partial A^{[2]}} \cdot \frac{\partial A^{[2]}}{\partial Z^{[2]}} \cdot \frac{\partial Z^{[2]}}{\partial W^{[2]}}$$

**Back-Propagation**



**Descenso de  
gradiente**



**Optimizadores**



# REDES NEURONALES — APRENDIZAJE

**Función de costo.** Una vez hemos llegado a la predicción final en la capa de salida, se calcula el error de predicción de la red (costo  $J$ ), basado en los errores individuales de cada predicción (loss  $L$ , función dependiente del tipo de función de activación y del contexto), para poder propagarlo a las capas anteriores:

$$J(\mathbf{W}^{[1]}, \mathbf{b}^{[1]}, \dots, \mathbf{W}^{[nCapas]}, \mathbf{b}^{[nCapas]}) = \frac{1}{m} \sum_{i=1}^m L(\hat{\mathbf{Y}}, \mathbf{Y})$$

Binary cross-entropy loss:  $L(\hat{\mathbf{y}}, \mathbf{y}) = -\frac{1}{N} \sum_i^N (y_i * \log(\hat{y}_i) + (1 - y_i) * \log(1 - \hat{y}_i))$

Categorical cross-entropy loss:  $L(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_i^N p(y_i) * \log(p(\hat{y}_i))$

Mean square loss:  $L(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{N} \sum_i^N (\hat{y}_i - y_i)^2$

Mean square logarithmic loss:  $L(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{N} \sum_i^N (\log(\hat{y}_i + 1) - \log(y_i + 1))^2$



# REDES NEURONALES — APRENDIZAJE

**Feed Forward para múltiples registros.** El mismo proceso se generaliza:

- En vez de tener un vector  $x^{(i)}$  con los valores de un registro, vamos a tener una matriz  $X$  con los vectores columnares del conjunto de instancias de aprendizaje, donde el número de columnas es  $m$  (el número de registros) y el número de filas es el número de inputs.
- La primera capa de neuronas pasa de producir un vector  $z^{[1](i)}$  para cada registro  $i$  a producir una matriz  $Z^{[1]} = W^{[1]}X + b^{[1]}$  para todos los registros.
- Igualmente pasamos de un vector de activación  $a^{[1](i)}$  para cada registro a una matriz  $A^{[1]} = g(Z^{[1]})$
- Y así sucesivamente para cada capa siguiente. Dada una capa  $[l]$  con  $n$  neuronas y  $k$  de entrada:

$$\begin{array}{c}
 \xrightarrow{\text{instancias}} \\
 A^{[l]} = \begin{bmatrix} a_1^{[l](1)} & a_1^{[l](2)} & \dots & a_1^{[l](m)} \\ a_2^{[l](1)} & a_2^{[l](2)} & \dots & a_2^{[l](m)} \\ \vdots & \vdots & \ddots & \vdots \\ a_n^{[l](1)} & a_n^{[l](2)} & \dots & a_n^{[l](m)} \end{bmatrix} \quad \begin{array}{c} \text{Neuronas de la capa L} \end{array} \\
 \downarrow
 \end{array}
 = g(Z^{[l]}) = g \left( \begin{array}{c} \text{Neuronas de la capa L-1} \\ \begin{bmatrix} w_{11}^{[l]} & w_{12}^{[l]} & \dots & w_{1k}^{[l]} \\ w_{21}^{[l]} & w_{22}^{[l]} & \dots & w_{2k}^{[l]} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1}^{[l]} & w_{n2}^{[l]} & \dots & w_{nk}^{[l]} \end{bmatrix} \begin{array}{c} \xrightarrow{\text{instancias}} \\ \begin{bmatrix} a_1^{[l-1](1)} & a_1^{[l-1](2)} & \dots & a_1^{[l-1](m)} \\ a_2^{[l-1](1)} & a_2^{[l-1](2)} & \dots & a_2^{[l-1](m)} \\ \vdots & \vdots & \ddots & \vdots \\ a_k^{[l-1](1)} & a_k^{[l-1](2)} & \dots & a_k^{[l-1](m)} \end{bmatrix} \end{array} \right) + \begin{bmatrix} b_1^{[l]} \\ b_2^{[l]} \\ \vdots \\ b_n^{[l]} \end{bmatrix}
 \end{array}$$



# REDES NEURONALES — APRENDIZAJE

## Back-propagation.

Para aprender los parámetros utilizamos descenso de gradiente, calculando entonces las derivadas parciales de la función de costo con respecto a cada parámetro de cada capa.

Se empieza por una inicialización aleatoria de los valores de los parámetros, que se irán actualizando a través de varias iteraciones según una tasa de aprendizaje  $\alpha$  aplicada al gradiente correspondiente:

$$W^{[l]} := W^{[l]} - \alpha \frac{\partial J}{\partial W^{[l]}} \quad b^{[l]} := b^{[l]} - \alpha \frac{\partial J}{\partial b^{[l]}}$$

Todo va a depender de las funciones de activación y sus derivadas parciales, veamos cuales serían los gradientes.

Finalmente, se compara el resultado del modelo con la etiqueta real utilizando una **función de pérdida**, que dependerá del tipo de modelo y del tipo de salida (logits, probabilidades, predicción numérica). Para clasificación se utilizan binary cross-entropy, categorical cross-entropy, negative log likelihood (NLL). Para regresión se utiliza MSE sobretodo.



# REDES NEURONALES — APRENDIZAJE

**Ejemplo para neuronas logísticas.** En el caso de una función de activación sigmoide  $g(Z) = \sigma(Z)$ , el descenso de gradiente se hace a partir de la derivada  $\sigma'(Z)$

$$\sigma(Z) = \frac{1}{1+e^{-Z}} = (1 + e^{-Z})^{-1}$$

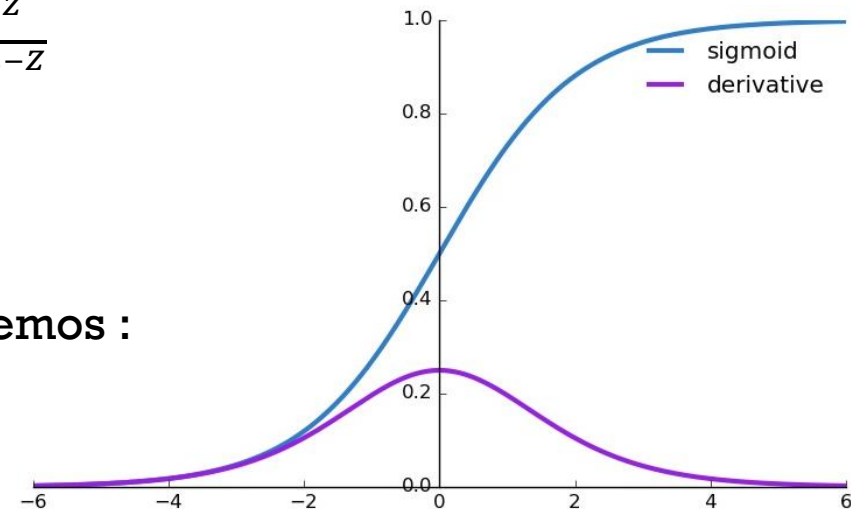
**Demostrar que  $\sigma'(Z) = \sigma(Z) \cdot (1 - \sigma(Z))$**

$$\begin{aligned} \sigma'(Z) &= -(1 + e^{-Z})^{-2}(-e^{-Z}) = \frac{e^{-Z}}{(1+e^{-Z})^2} = \frac{1}{1+e^{-Z}} \cdot \frac{e^{-Z}}{1+e^{-Z}} \\ &= \frac{1}{1+e^{-Z}} \cdot \frac{1+e^{-Z}-1}{1+e^{-Z}} = \frac{1}{1+e^{-Z}} \cdot \left(1 - \frac{1}{1+e^{-Z}}\right) \\ &= \sigma(Z) \cdot (1 - \sigma(Z)) \end{aligned}$$

Para las activaciones sigmoides de una capa  $l$ , tenemos :

$$A^{[l]} = g(Z^{[l]}) = \sigma(Z^{[l]})$$

$$A'^{[l]} = A^{[l]} \cdot (1 - A^{[l]})$$



→ Para hacer el back-propagation, vamos a guardar las activaciones calculadas en el feed-forward

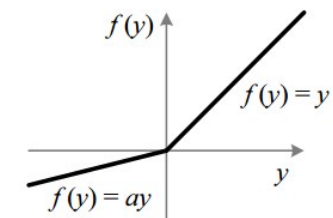
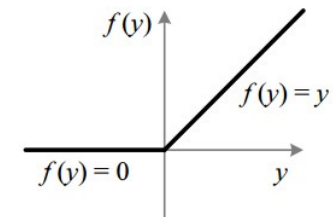
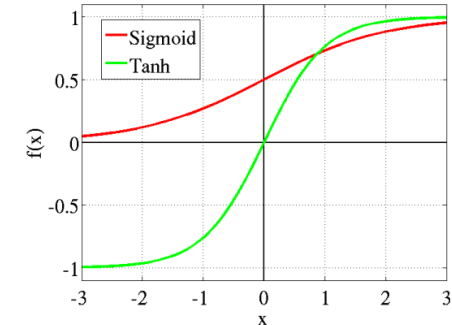


# REDES NEURONALES — APRENDIZAJE

## Gradientes para las activaciones más comunes en DL

Encontramos que para las funciones de activación más comunes, los gradientes se definen en función del cálculo de las activaciones previamente computados durante la fase de feed-forward:

- Sigmoide.  $a(Z) = \frac{1}{1+e^{-Z}}$   $a'(Z) = a(Z) \cdot (1 - a(Z))$
- Tanh.  $a(Z) = \frac{e^Z - e^{-Z}}{e^Z + e^{-Z}}$   $a'(Z) = 1 - a^2(Z)$
- ReLU.  $a(Z) = \max(0, Z)$   $a'(Z) = \begin{cases} 0, & \text{si } Z < 0 \\ 1, & \text{si } Z > 0 \\ \text{indefinido}, & \text{si } Z = 0 \end{cases}$
- Leaky ReLU.  $a(Z) = \max(0.01 * Z, Z)$   $a'(Z) = \begin{cases} 0.01, & \text{si } Z < 0 \\ 1, & \text{si } Z > 0 \\ \text{indefinido}, & \text{si } Z = 0 \end{cases}$



[www.towardsdatascience.com](http://www.towardsdatascience.com)



# REDES NEURONALES — APRENDIZAJE

**Back-propagation.** Ilustremos el proceso con un caso de clasificación binaria con una neurona sigmoide en la capa final y una sola capa escondida.

$$\begin{array}{l}
 A^{[1]} \rightarrow \\
 W^{[2]} \rightarrow \\
 b^{[2]} \rightarrow
 \end{array}
 \begin{array}{l}
 \rightarrow \\
 \rightarrow \\
 \rightarrow
 \end{array}
 Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]} \longrightarrow A^{[2]} = \sigma(Z^{[2]}) \longrightarrow$$

$$\begin{array}{l}
 \frac{\partial Z^{[2]}}{\partial b^{[2]}} = 1 \quad \frac{\partial Z^{[2]}}{\partial W^{[2]}} = A^{[1]} \quad \frac{\partial A^{[2]}}{\partial Z^{[2]}} = \sigma'(Z^{[2]}) = \sigma(Z^{[2]}) (1 - \sigma(Z^{[2]}))
 \end{array}$$

$$\begin{array}{l}
 L(\hat{Y}, Y) = L(A^{[2]}, Y) = -Y * \log(A^{[2]}) - (1 - Y) * \log(1 - A^{[2]}) \\
 \frac{\partial L}{\partial A^{[2]}} = -\frac{Y}{A^{[2]}} + \frac{1 - Y}{1 - A^{[2]}}
 \end{array}$$

Durante la fase feed forward se calculan los valores intermedios de  $A^{[1]}$ ,  $Z^{[2]}$  y  $A^{[2]}$ , que aplicados a las derivadas parciales y gracias a la regla de la cadena permiten encontrar los valores del gradiente de la función de pérdida utilizados para la actualización de los parámetros de la capa  $W^{[2]}$  y  $b^{[2]}$ .

$$\left. \begin{array}{l}
 \frac{\partial L}{\partial W^{[2]}} = \frac{\partial L}{\partial A^{[2]}} \cdot \frac{\partial A^{[2]}}{\partial Z^{[2]}} \cdot \frac{\partial Z^{[2]}}{\partial W^{[2]}} \\
 \frac{\partial L}{\partial b^{[2]}} = \frac{\partial L}{\partial A^{[2]}} \cdot \frac{\partial A^{[2]}}{\partial Z^{[2]}} \cdot \frac{\partial Z^{[2]}}{\partial b^{[2]}}
 \end{array} \right\}$$



# REDES NEURONALES — APRENDIZAJE

**Back-propagation.** Ilustremos el proceso con un caso de clasificación binaria con una neurona sigmoide en la capa final, un solo registro y una sola capa escondida (parámetros  $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}$ ).

$$\begin{aligned}
 & \mathbf{a}^{[1]} \rightarrow \mathbf{z}^{[2]} = \mathbf{W}^{[2]} \mathbf{a}^{[1]} + \mathbf{b}^{[2]} \rightarrow \mathbf{a}^{[2]} = \sigma(\mathbf{z}^{[2]}) \rightarrow L(\hat{y}, y) = L(\mathbf{a}^{[2]}, y) = -y * \log(\mathbf{a}^{[2]}) - (1 - y) * \log(1 - \mathbf{a}^{[2]}) \\
 & \frac{\partial L}{\partial \mathbf{W}^{[2]}} = \frac{\partial L}{\partial \mathbf{z}^{[2]}} \cdot \frac{\partial \mathbf{z}^{[2]}}{\partial \mathbf{W}^{[2]}} = (\mathbf{a}^{[2]} - y) \mathbf{a}^{[1]} \quad \frac{\partial \mathbf{a}^{[2]}}{\partial \mathbf{z}^{[2]}} = \sigma'(\mathbf{z}^{[2]}) = \mathbf{a}^{[2]}(1 - \mathbf{a}^{[2]}) \quad \frac{\partial L}{\partial \mathbf{a}^{[2]}} = -\frac{y}{\mathbf{a}^{[2]}} + \frac{1 - y}{1 - \mathbf{a}^{[2]}} = \frac{\mathbf{a}^{[2]} - y}{\mathbf{a}^{[2]}(1 - \mathbf{a}^{[2]})} \\
 & \frac{\partial L}{\partial \mathbf{b}^{[2]}} = \frac{\partial L}{\partial \mathbf{z}^{[2]}} \cdot \frac{\partial \mathbf{z}^{[2]}}{\partial \mathbf{b}^{[2]}} = (\mathbf{a}^{[2]} - y) \quad \frac{\partial L}{\partial \mathbf{z}^{[2]}} = \frac{\partial L}{\partial \mathbf{a}^{[2]}} \cdot \frac{\partial \mathbf{a}^{[2]}}{\partial \mathbf{z}^{[2]}} = \mathbf{a}^{[2]} - y
 \end{aligned}$$

El mismo proceso se realiza para la capa intermedia, donde  $\mathbf{A}^{[0]} = \mathbf{X}$ .

$$\frac{\partial L}{\partial \mathbf{z}^{[1]}} = \frac{\partial L}{\partial \mathbf{a}^{[2]}} \cdot \frac{\partial \mathbf{a}^{[2]}}{\partial \mathbf{z}^{[2]}} \cdot \frac{\partial \mathbf{z}^{[2]}}{\partial \mathbf{a}^{[1]}} \cdot \frac{\partial \mathbf{a}^{[1]}}{\partial \mathbf{z}^{[1]}} = (\mathbf{a}^{[2]} - y) \cdot \mathbf{W}^{[2]} \cdot \sigma'(\mathbf{z}^{[1]})$$



# REDES NEURONALES — APRENDIZAJE

- Un resultado importante en cuanto al gradiente de la **última capa** con respecto a la combinación lineal  $Z$  entrante es que no es necesario considerar el gradiente de la función de activación si está es la función sigmoide o softmax, pues tenemos directamente:

$$\frac{\partial L}{\partial z^{[N]}} = \frac{\partial L}{\partial a^{[N]}} \cdot \frac{\partial a^{[N]}}{\partial z^{[N]}} = a^{[N]} - y = \hat{y} - y$$



# TALLER ANN: BACKPROP DE XOR CON NUMPY

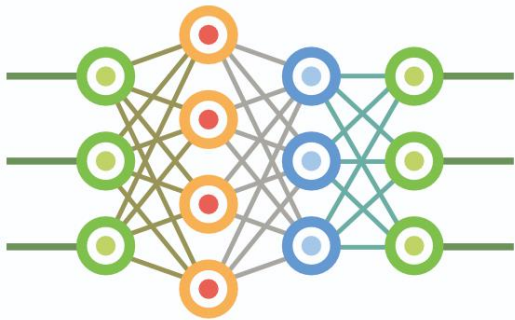
Para el problema de XOR con una sencilla red de una capa con 3 entradas, una capa escondida con 4 neuronas y una capa con 1 neurona de salida, utilizando funciones sigmoides y sin considerar sesgos en las neuronas ( $b = 0$ ):

- Desarrollar la función **backProp(X, y, w1, w2, b1, b2, a2, a1)** que calcula el vector con las predicciones para el conjunto de registros de entrada. Retorna las activaciones de cada capa para los pesos y sesgos de la capa escondida ( $dw1$ ,  $db1$ ) y de la capa de salida ( $dw2$ ,  $db2$ )
- Desarrollar la función **entrenarRed(epocas, lr, X, y, w1, b1, w2, b2, a1, a2)** del ciclo de entrenamiento para un número específico de épocas, utilizando una tasa de aprendizaje definida.

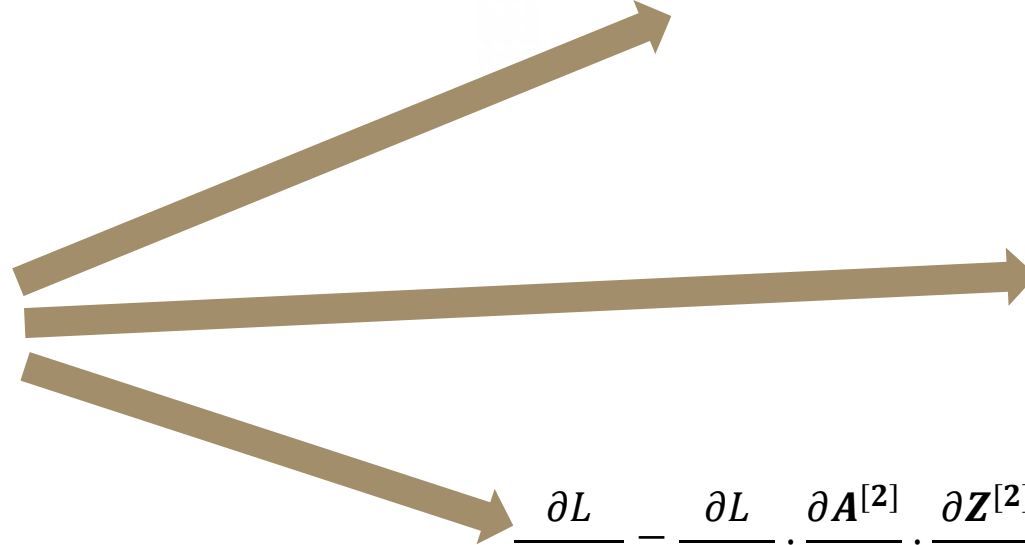


# AGENDA

 PyTorch

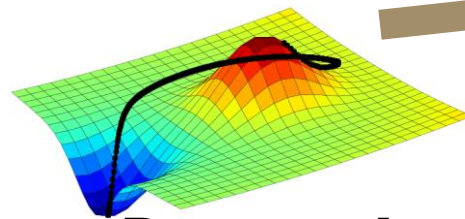


**Redes Neuronales  
Artificial (ANN)**

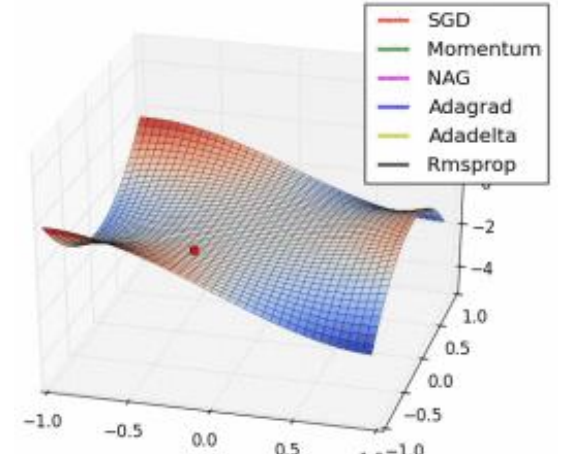


$$\frac{\partial L}{\partial W^{[2]}} = \frac{\partial L}{\partial A^{[2]}} \cdot \frac{\partial A^{[2]}}{\partial Z^{[2]}} \cdot \frac{\partial Z^{[2]}}{\partial W^{[2]}}$$

**Back-Propagation**



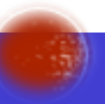
**Descenso de  
gradiente**



**Optimizadores**



# OPTIMIZADORES



# OPTIMIZADORES

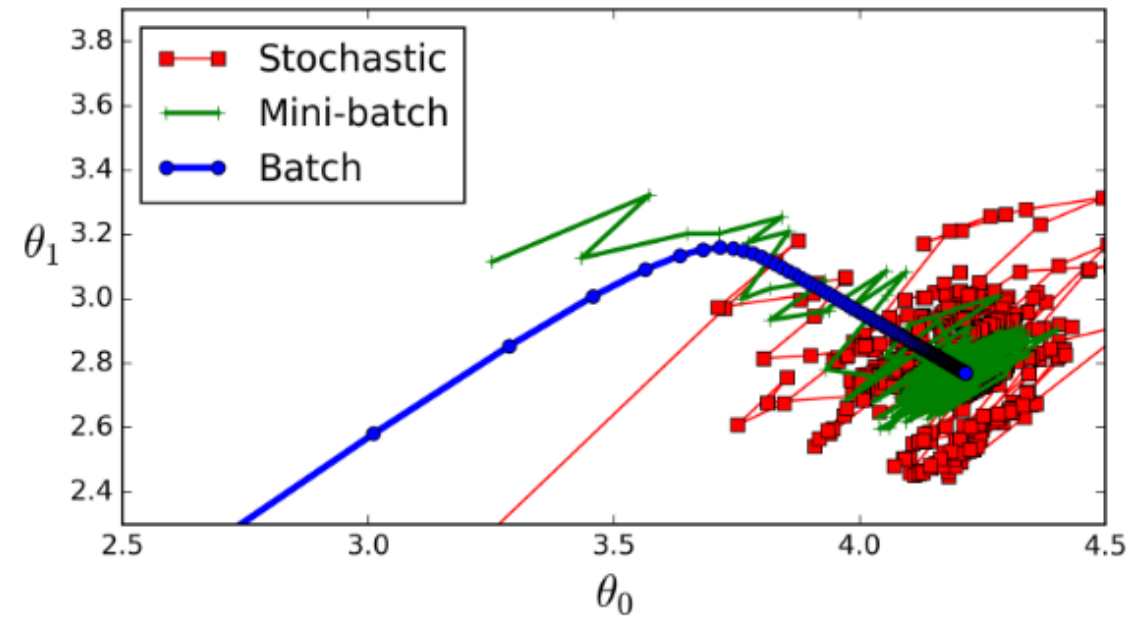
- En DL como en ML, el **aprendizaje** consiste en encontrar los valores de los parámetros del modelo que minimicen una función de costo dada.
- La **función de costo** debe estar íntimamente ligada a la solución del problema, de tal manera que entre más pequeño el costo, mejor el modelo representa el mapeo entre inputs y outputs deseados.
- Se utilizan algoritmos **optimizadores** para minimizar la función de costo. Todos se basan en los principios del **descenso de gradiente**, que en su forma pura (**batch gradient descent**) actualiza los parámetros solo después de haber calculado el costo para la totalidad del conjunto de aprendizaje (una época). El problema de este optimizador es que, aunque válido, suele ser muy lento (c.f. Big Data). Es por eso que los frameworks como TF incluyen alternativas buscando una convergencia mas rápida.





# OPTIMIZADORES

- **Mini batch gradient descent:** se calcula el costo y sus derivadas parciales con respecto a los parámetros (delta de actualización) con una fracción del dataset, de tal manera que en cada época se consideran varios batches, produciendo cada uno una actualización de los parámetros. Para optimizar la paralelización se consideran batches de tamaños en las potencias de 2.
- **Stochastic gradient descent:** como un mini batch con tamaño 1; se escogen registros al azar y se actualizan los parámetros a partir de la propagación del único costo de su predicción.



*Gradient Descent paths in parameter space*

Géron, 2017

→ Se cambia la certeza de la mejora segura de la función de costo en cada paso (varianza) por la escalabilidad en cuanto al tamaño del dataset y del modelo



# TALLER ANN: MNIST CON NUMPY

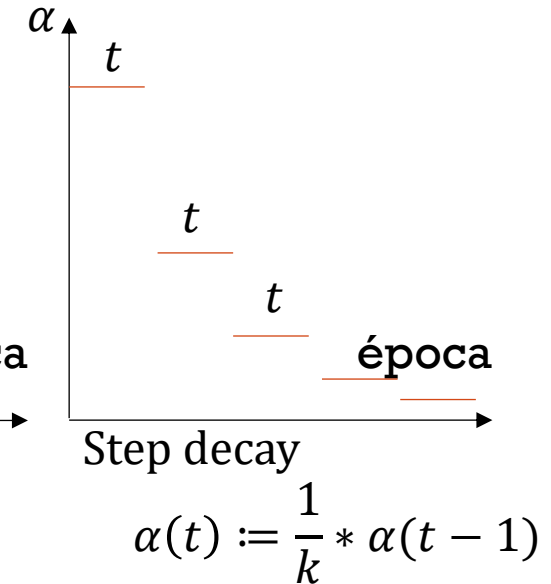
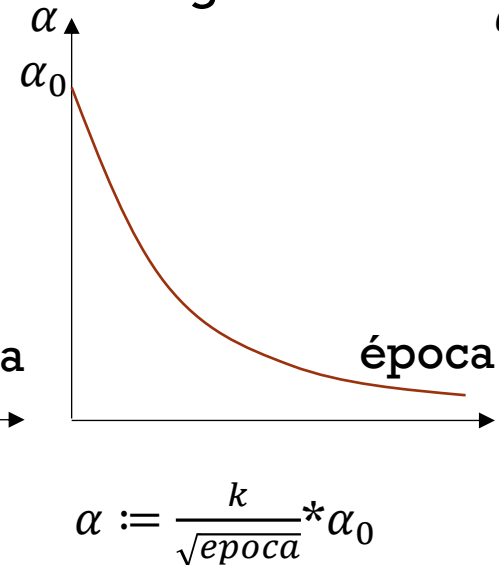
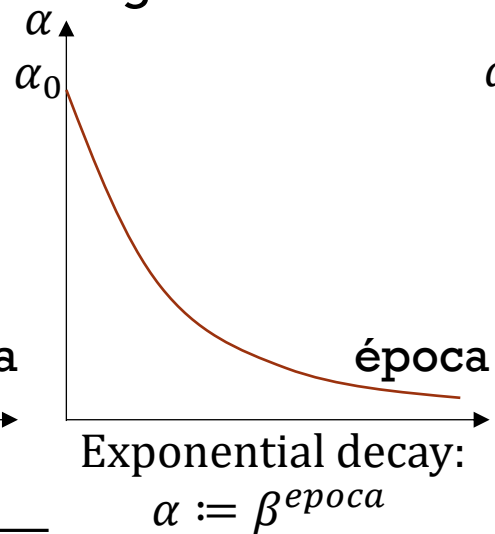
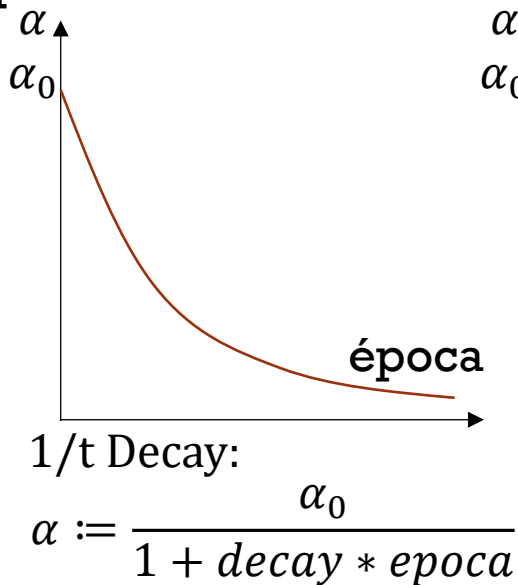
Para el problema de MNIST se define una red neuronal con 1 capa de entrada de  $28 \times 28 = 784$  neuronas de entrada, 1 capa escondida de 512 neuronas y una capa de salida de 10 neuronas con una función de activación softmax. Completar el código faltante, teniendo en cuenta para que funcione con una función de activación ReLU o tanh (dejar el código para tanh y dejar en comentario las líneas de ReLU en las funciones **feedForward** y **backProp**. Completar los códigos de las funciones:

- funciones **softmax** (no es necesaria la función gradiente, pues softmax solo se puede utilizar en la capa de salida, y generaliza la simplificación de softmax), **tanh**, **tanhGrad**, **relu**, **reluGrad**
- **initParams()**
- **costoGlobal(y\_est, y)**
- **feedForward(X, w1, b1, w2, b2)**
- **backProp(X, y, w1, b1, w2, b2, a1, a2)**
- **entrenarRed(épocas, lr, X, y, w1, b1, w2, b2, a1, a2)**
- **entrenarRedMiniBatch(épocas, batch\_size, lr, X, y, w1, b1, w2, b2, a1, a2)**



# OPTIMIZADORES

- **Degradación de la tasa de aprendizaje:** la idea es no utilizar siempre la misma tasa de aprendizaje, pues para las épocas más avanzadas, puede que esta sea demasiado elevada y no permita llegar a convergencia. Se propone entonces aplicar una función de degradación como las siguientes:





# OPTIMIZADORES

- **Adagrad:** Adapta la tasa de aprendizaje a los parámetros, aplicando menores actualizaciones a parámetros asociados a valores más comunes y mayores actualizaciones a parámetros asociados a valores menos frecuentes
- **AdaDelta:** extiende AdaGrad, usando una ventana móvil de gradientes, en vez de considerar todos los gradientes, para evitar el decrecimiento del learning rate. Se aplica en problemas con datos dispersos (tratamiento de textos, sistemas de recomendación)

$$S(\theta_i) := S(\theta_i) + \left( \frac{\partial}{\partial \theta_i} J(\Theta) \right)^2$$
$$\theta_i := \theta_i - \alpha * \frac{\left( \frac{\partial}{\partial \theta_i} J(\Theta) \right)^2}{\sqrt{S(\theta_i)} + \epsilon}$$



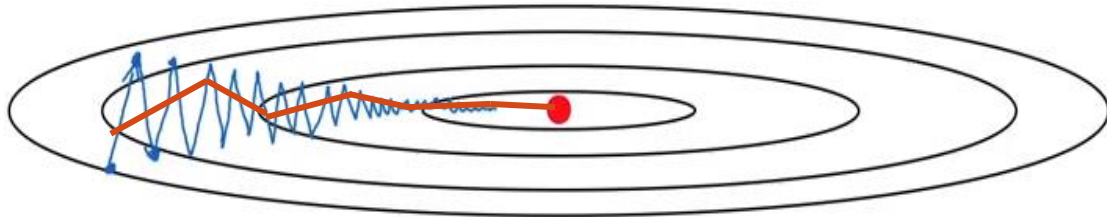
# OPTIMIZADORES

## ▪ Momentum:

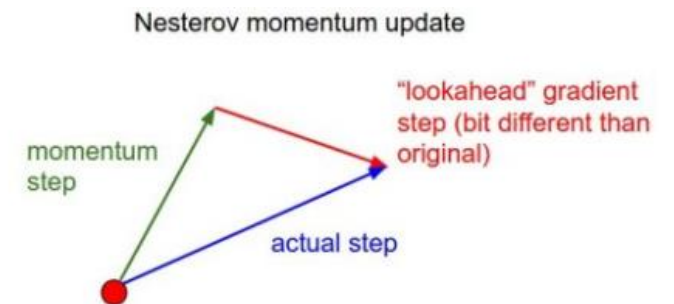
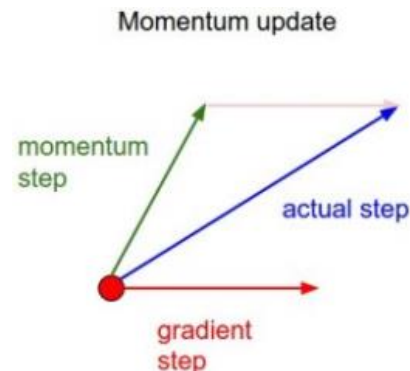
- Puede que el gradiente con respecto a uno de los parámetros sea más importante que con respecto a otros y contrario a la dirección de mayor minimización del costo
- la idea es utilizar no solo el gradiente del paso actual, sino los de los últimos pasos, haciendo un promedio ponderado exponencial (usualmente  $\beta = 0.9$ ):

$$\text{Momentum}(\theta_i) := \beta * \text{Momentum}(\theta_i) + (1 - \beta) \frac{\partial}{\partial \theta_i} J(\Theta) \quad \theta_i := \theta_i - \alpha * \text{Momentum}(\Theta)$$

- Nesterov: promediar con el gradiente del momento



Andrew Ng, Coursera, 2017



# OPTIMIZADORES

- **RMSPprop**: (Root Mean Square Propagation) considera un factor de corrección de la regla de actualización del gradient descent normal según el inverso del promedio exponencial ponderado de los cuadrados de las derivadas parciales con respecto a los parámetros:

$$S(\theta_i) := \beta * S(\theta_i) + (1 - \beta) \left( \frac{\partial}{\partial \theta_i} J(\Theta) \right)^2 \quad \theta_i := \theta_i - \alpha * \frac{\partial}{\partial \theta_i} J(\Theta) * \frac{1}{\sqrt{S(\theta_i) + \epsilon}}$$

- Lo que se busca es que se priorice la dirección de actualización que implique la menor reducción del costo (misma idea que con el momentum)
- Se puede entonces con estos métodos utilizar una tasa de aprendizaje mayor alejando la posibilidad de divergencia
- Se ha aplicado exitosamente a redes recurrentes



# OPTIMIZADORES

- **Adam:** (Adaptive Moment Estimation) Empíricamente se ha aplicado existosamente para aprender modelos con muchos tipos de arquitecturas diferentes con objetivos diferentes. Conserva un promedio con decaimiento exponencial tanto de los gradientes pasados (Momentum) como de los gradientes al cuadrado (RMSProp)

$$Momentum(\theta_i) := \beta_1 * Momentum(\theta_i) + (1 - \beta_1) \frac{\partial}{\partial \theta_i} J(\Theta)$$

$$S(\theta_i) := \beta_2 * S(\theta_i) + (1 - \beta_2) \left( \frac{\partial}{\partial \theta_i} J(\Theta) \right)^2$$

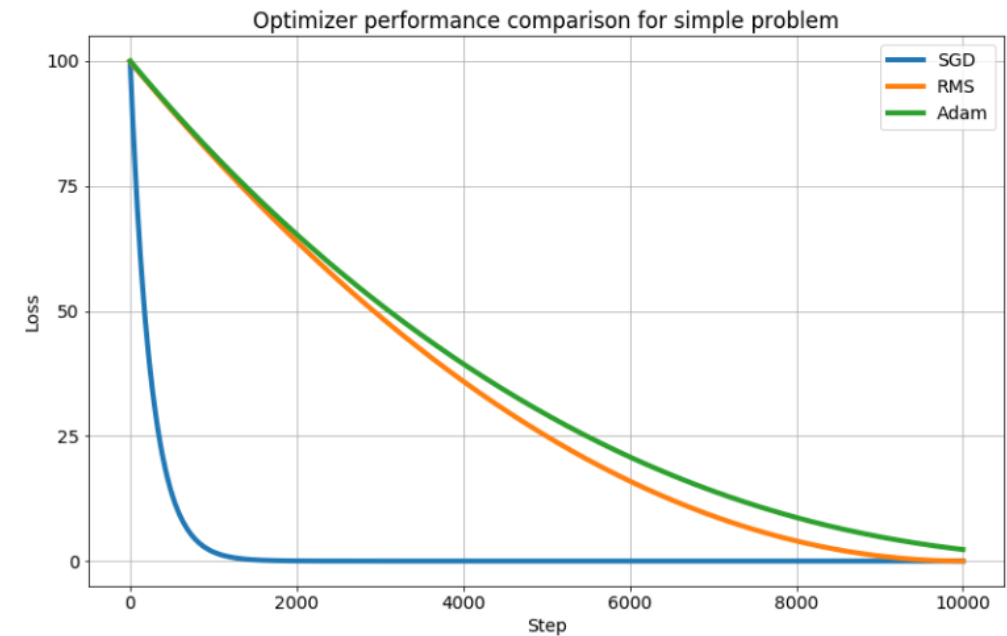
$$\theta_i := \theta_i - \alpha * Momentum(\theta_i) * \frac{1}{\sqrt{S(\theta_i)} + \epsilon}$$

- Los valores de los hiperparámetros  $\beta_1$  y  $\beta_2$  utilizados por defecto son 0.9 y 0.99 respectivamente
- Usualmente supera a todos los demás optimizadores



# OPTIMIZADORES

- Para problemas sencillos, SGD puede llegar a ser una buena alternativa, requiriendo menos épocas de entrenamiento



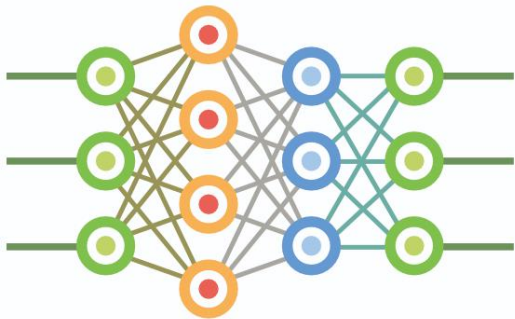
# OVERFITTING

- Como cualquier modelo de ML, las redes neuronales (shallow o deep) pueden sufrir de overfitting, cuando:
  - La cantidad de datos es poca
  - La arquitectura es excesivamente compleja (número de capas, número de neuronas, etc.)
  - El número de épocas de aprendizaje es excesivo
- Hay varias maneras de combatir el overfitting
  - Adquirir más datos reales, o crear datos nuevos basados en reales (**data augmentation**)
  - Reutilizar partes de la arquitectura previamente entrenadas con datasets relacionados de mayor volumen (**transfer learning**)
  - Monitorear el desempeño en un dataset de validación durante el entrenamiento del modelo, estableciendo en qué época se optimiza (**early stopping**)
  - Utilizar técnicas de **regularización**: L1, L2, capas dropout, capas de pooling

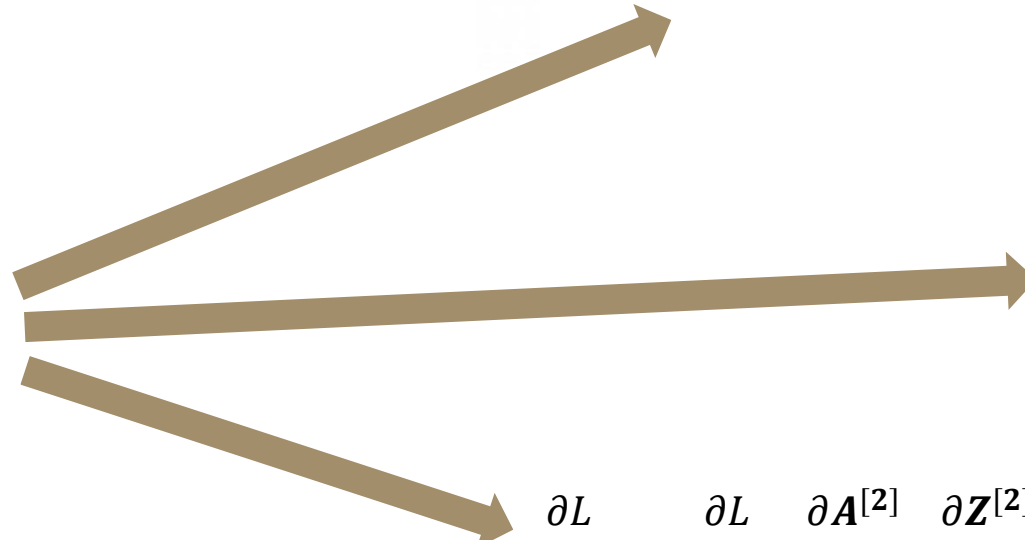


# AGENDA

 PyTorch

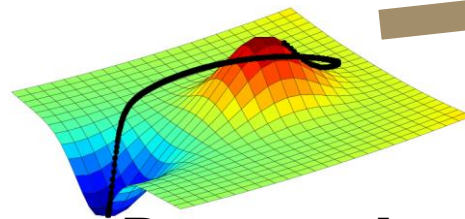


**Redes Neuronales  
Artificial (ANN)**

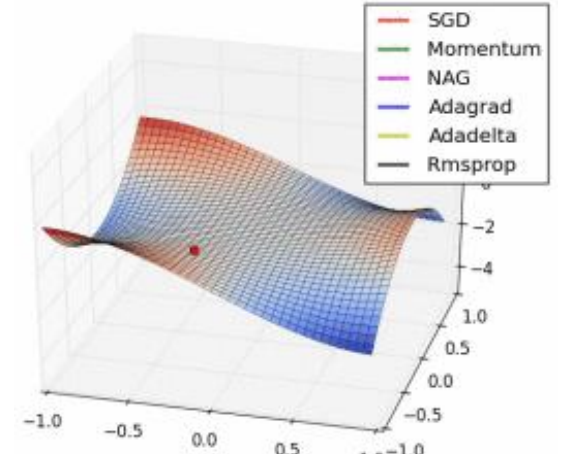


$$\frac{\partial L}{\partial W^{[2]}} = \frac{\partial L}{\partial A^{[2]}} \cdot \frac{\partial A^{[2]}}{\partial Z^{[2]}} \cdot \frac{\partial Z^{[2]}}{\partial W^{[2]}}$$

**Back-Propagation**



**Descenso de  
gradiente**



**Optimizadores**



# FRAMEWORKS DE DL

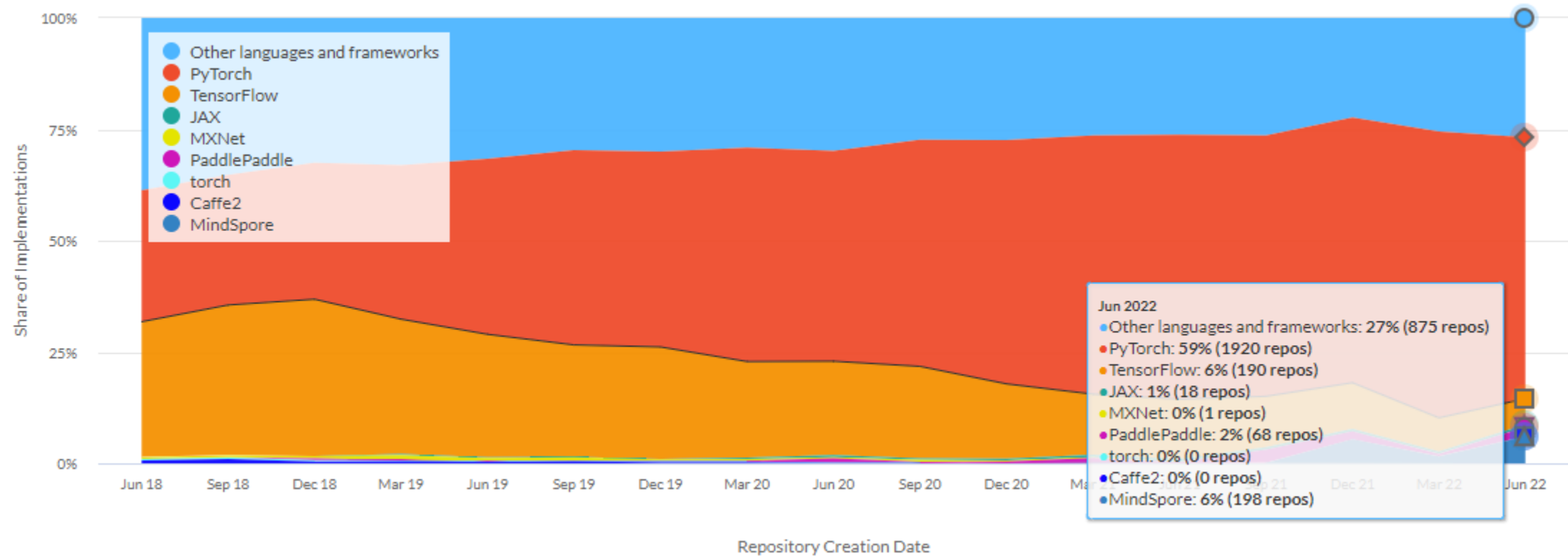
theano mxnet

 TensorFlow Microsoft  
CNTK

Caffe K Keras

 PyTorch DL4J Caffe2 PyTorch  
Lightning






<https://paperswithcode.com/trends>, June 2022




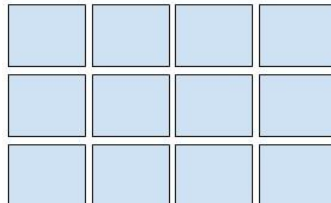
# TENSORES

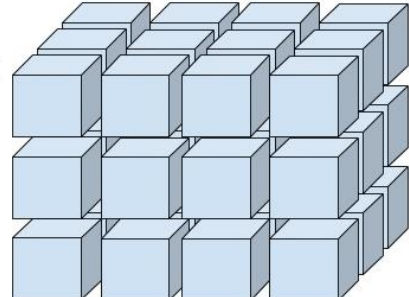
- Los datos se representan en forma de tensores: arrays multi dimensionales
- El **rango** de un tensor es el número de ejes que este tiene, cada rango tiene un número de dimensiones determinado
- Por convención,
  - los **escalares** son tensores de rango 0
  - el **primer eje** siempre será utilizado para separar las instancias, independientemente del tipo de datos (vectores de features, secuencias, imágenes, videos, etc.)
  - Para los datos **vectoriales** (hojas de cálculo clásicas), se tiene que la primera dimensión se usa para las instancias y la segunda para los features

- Al realizar operaciones elemento por elemento sobre 2 tensores, cuando no se tienen el mismo número de dimensiones, numpy realiza la operación de **broadcasting**, que consiste en crear los ejes faltantes en el tensor más pequeño y repetir el contenido para rellenarlos.

Rank 0:   
(scalar)

Rank 1:   
(vector)

Rank 2: (matrix)  


Rank 3: 

Raschka, 2017

Una hoja de Excel con 10 columnas que tipo de tensor es?  
Y un set de imágenes RGB? Y un video basado en imágenes RGB?



# TENSORES

- Por convención,
  - Para las **secuencias** o **series temporales** cada instancia tiene la foto de varios features en varios momentos de tiempo. Se tiene como convención, en el caso de varias instancias de fotos temporales los siguientes ejes:
    - el primer eje es el de las instancias, como ya se había aclarado
    - el segundo eje se utiliza para la temporalidad
    - el tercer eje contiene las features tomadas del instante dado por el segundo eje
  - Para las **imágenes**, al tener un eje para el canal de color, hay dos convenciones diferentes:
    - **channels-first**: primer eje para instancias, segundo para los canales de color, tercero y cuarto para alto y ancho (el más utilizado por **PyTorch**)
    - **channels-last**: primer eje para instancias, segundo y tercero para alto y ancho, cuarto para los canales de color (el más utilizado por **TensorFlow**).
  - Los **videos** siguen las mismas dos convenciones de las imágenes, teniendo en cuenta que cada frame es una imagen. En el caso de **channels-first**, se tendrían entonces los ejes arreglados de la siguiente manera: instancias, frames, color, alto, ancho.



# PYTORCH



- Framework open source, desarrollado por Meta AI (Facebook) desde 2016, brindando dos grandes funcionalidades de alto nivel:
  - Computación a base de tensores, con aceleración en base a cálculos sobre GPUs
  - Sistema de diferenciación automática sobre grafos computacionales dinámicos (no necesitan compilación), base de modelos de Deep Learning
- Flexibilidad: no es necesario encargarse de detalles de bajo nivel (e.g. la codificación de los gradientes)
- Interfaz con GPUs para procesamiento paralelo, sin necesidad de ocuparse de NINGUN detalle de programación de bajo nivel.
  - Si bien la base es sobre tarjetas Nvidia (CUDA), también hay implementaciones para AMD RocM y Apple.
  - Se cargan los datos y parámetros en la memoria de GPU
- Permite distribuir la carga en múltiples dispositivos (e.g. varios CPUs y/o GPUs)

<https://pytorch.org>



# PYTORCH



- Entre las utilidades de PyTorch, las mas usadas son:
  - Adicionalmente al paquete de **torch** (no “**pytorch**”), se instalan usualmente los paquetes **torchvision** y **torchaudio**
  - PyTorch se basa en componentes (las capas, funciones de activación y modelos) que heredan de **torch.nn.Module**, de tal manera que se pueden combinar en módulos mas grandes y complejos, interactuando entre ellos, realizando la propagación de gradientes internamente de manera automática.
  - Paquete **torch.nn**, que contiene los componentes para construir redes neuronales
    - **nn.Linear**: capas densas de combinación lineal de sus inputs, sin función de activación
    - **nn.ReLU**, **nn.Sigmoid**, **nn.Tanh**, **nn.softmax**: funciones de activación a aplicar al input recibido
    - **nn.Dropout**: capas de regularización dropout
    - **nn.BatchNorm[1D,2D,3D]**: capas de batch normalization para tensores de diferentes rangos
    - **nn.Conv[1D,2D,3D]**: capas convolucionales para tensores de diferentes rangos
    - **nn.MaxPool[1D,2D,3D]**: capas de pooling para tensores de diferentes rangos
    - **nn.CrossEntropyLoss**, **nn.BCELoss**, **nn.MSELoss**: funciones de pérdida

<https://pytorch.org>



# PYTORCH



- Entre las utilidades de PyTorch, las mas usadas son:
  - Paquete **torch.optim**,
    - Incluye clases con los algoritmos de optimización más utilizados como SGD, Adam, RMSProp, etc.
    - Se puede crear optimizadores propios, heredando de la clase **torch.optim.Optimizer**.
  - En el paquete **torchmetrics** se encuentran implementadas clases para mas de 80 métricas para evaluar la calidad de las predicciones de un modelo:
    - Métricas de clasificación: Accuracy, AUC, Kappa, precisión, recall, F1, especificidad, matriz de confusión, etc.
    - Métricas de regresión: MAE, MAPE, MSE, R2, correlación de Pearson, etc.
    - Métricas para audio, imágenes, detección, texto, ...
    - Es necesario instalar el paquete **torchmetrics** de manera independiente

<https://pytorch.org>



# TALLER INTRO A PYTORCH

Entender y ejecutar el taller introductorio de Pytorch.



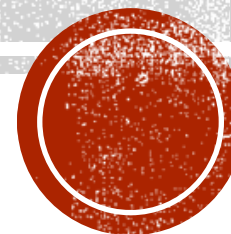
# TALLER MNIST EN PYTORCH

Desarrollar un clasificador del dataset MNIST utilizando Pytorch.





**GRACIAS**



# REFERENCIAS

- *Introduction to Statistical Learning with Applications in R (ISLR)*, G. James, D. Witten, T. Hastie & R. Tibshirani, 2014, Springer
- *Python Data Science Handbook*, Jake VanderPlas, 2017, O'Reilly
- *The Elements of Statistical Learning*, T. Hastie & R. Tibshirani, 2009, Springer
- *Python Machine Learning (2nd ed.)*, Sebastian Raschka, 2017, Packt
- *Learning TensorFlow*, Tom Hope, Yehezkel S. Resheff & Itay Lieder, O'Reilly 2017
- *Introduction to TensorFlow*, Chris Manning & Richard Socher, Lecture 7 of the CS224n course at Stanford University, 2017
- *Hands-On Machine Learning with Scikit-Learn and TensorFlow*, Aurélien Géron, 2017
- *TensorFlow for Deep Learning*, Charath Ramsundar & Reza Bosagh Zadeh, O'Reilly, 2018
- *Deep Learning with Python*, Francois Chollet, Manning 2018
- *Neural Networks and Deep Learning*, Andrew Ng, Coursera, 2017

