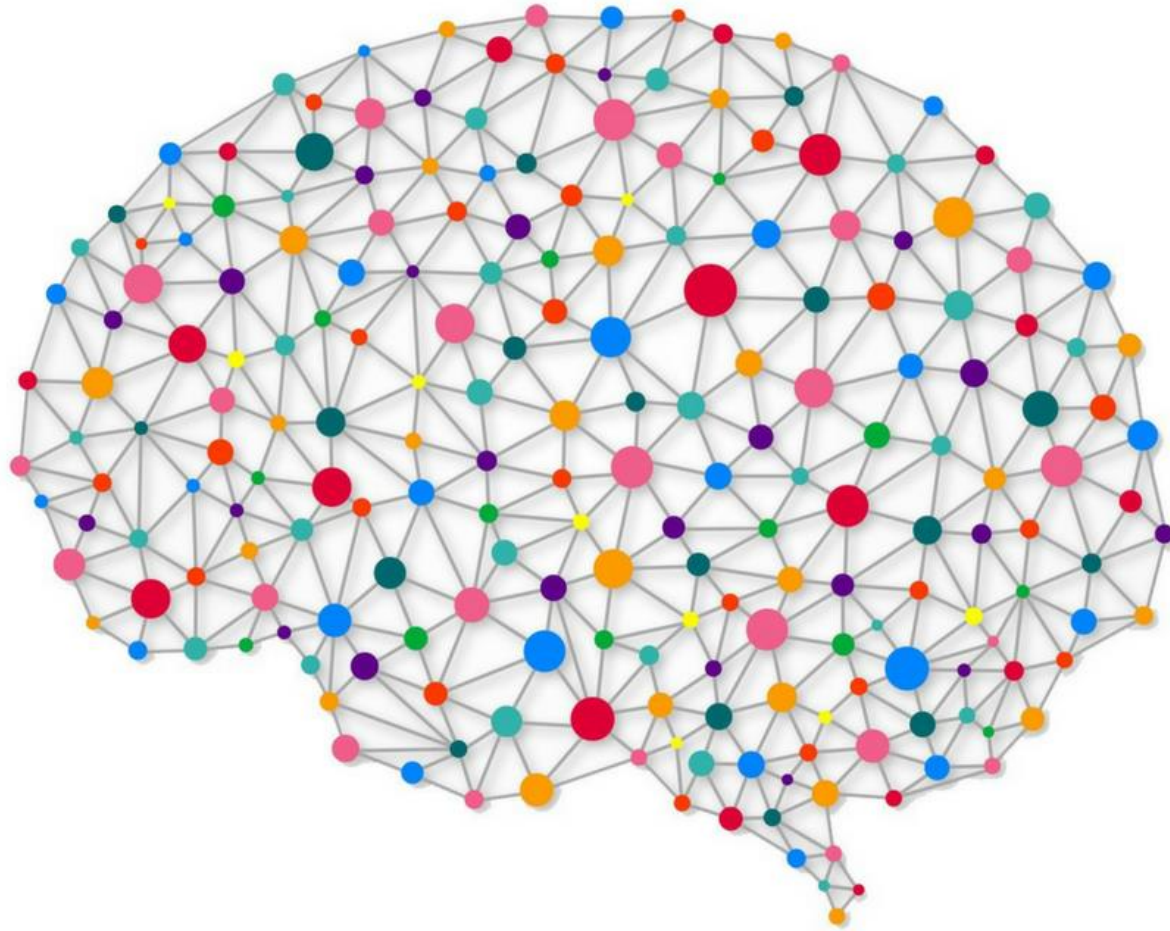


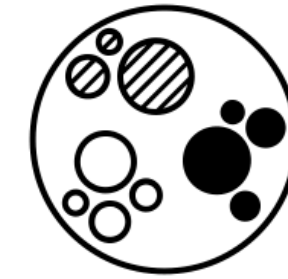
# TENSOR FLOW



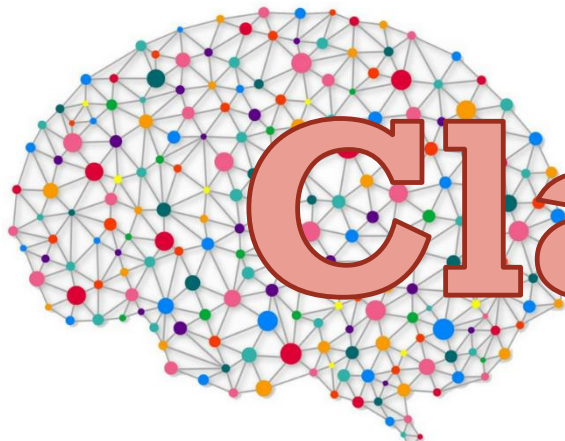
Javier Diaz Cely, PhD



# AGENDA



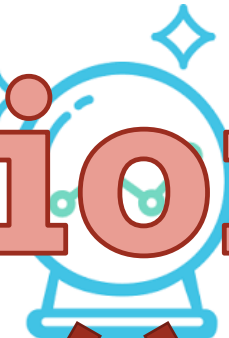
**Aprendizaje  
no supervisado**



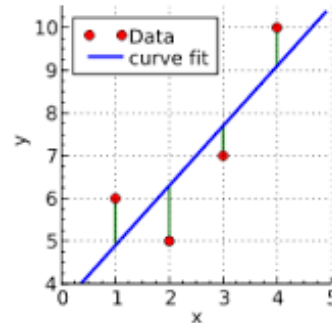
**Deep Learning**

# Clase anterior

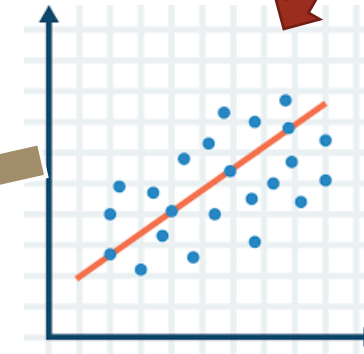
**Machine Learning  
(Aprendizaje  
Automático)**



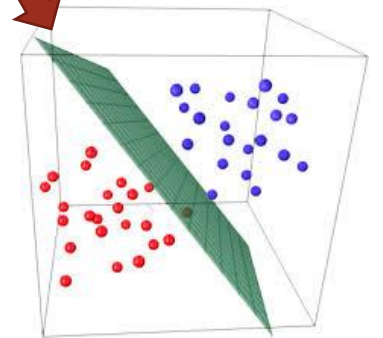
**Aprendizaje  
supervisado**



**Mínimos  
cuadrados  
ordinarios**



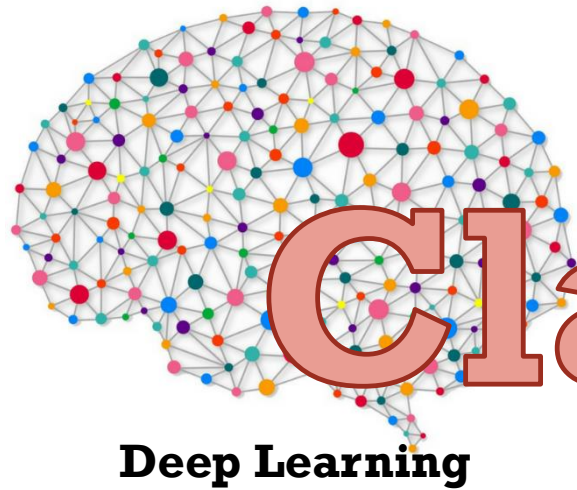
**Regresión**



**Clasificación**



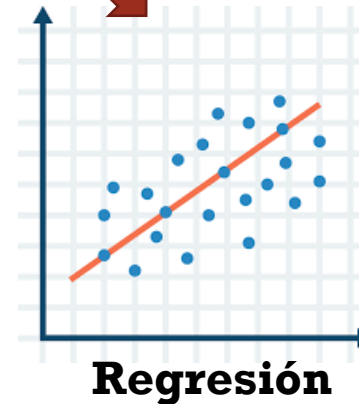
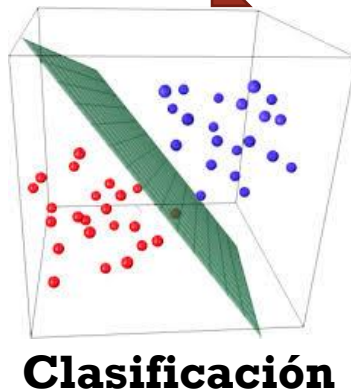
# AGENDA



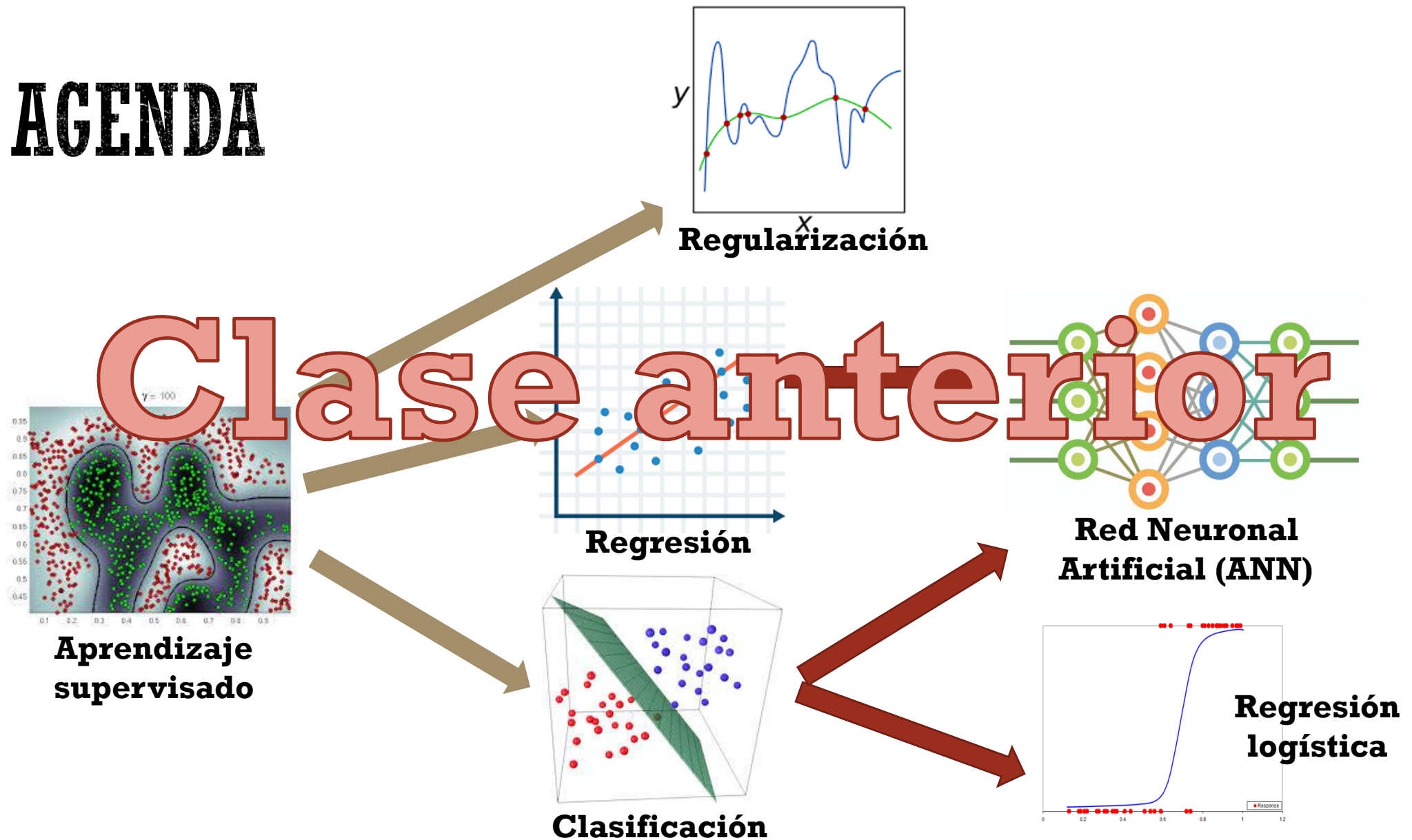
# Clase anterior

Aprendizaje  
supervisado

Descenso de  
gradiente



# AGENDA

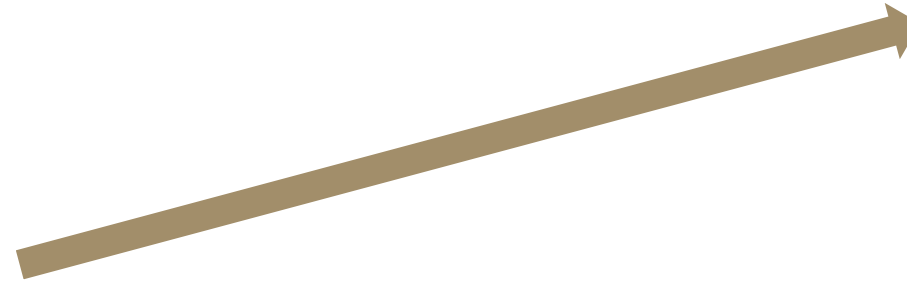




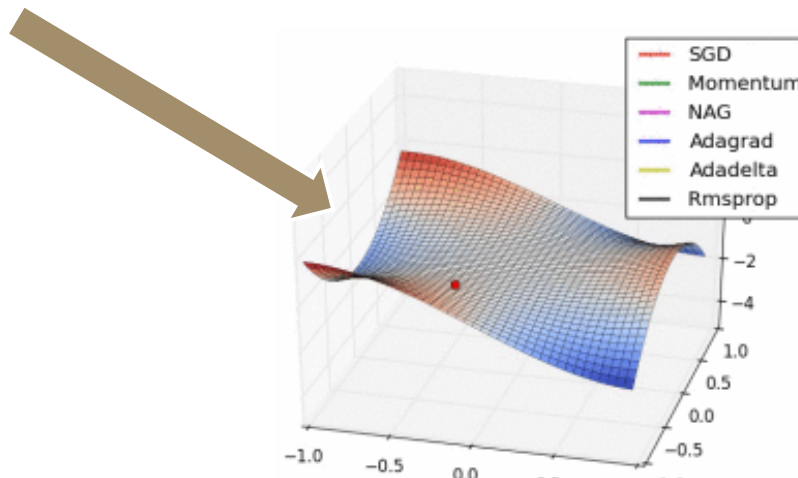
# AGENDA



**Deep Learning**



**Keras**



**Optimizadores**



# OPTIMIZADORES



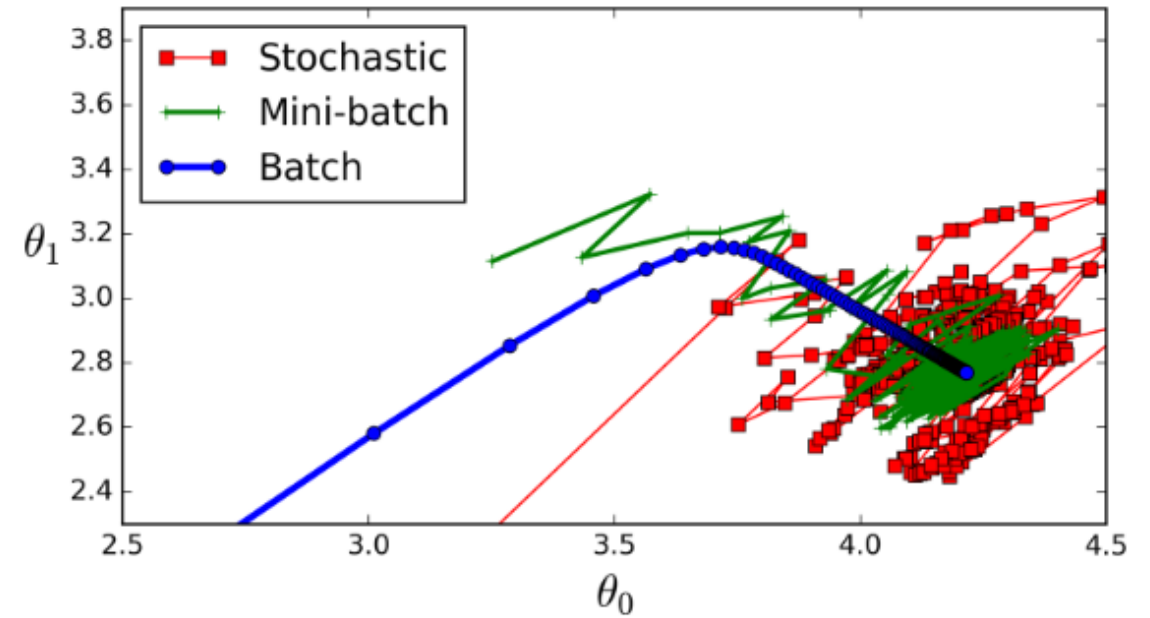
# OPTIMIZADORES

- En DL como en ML, el **aprendizaje** consiste en encontrar los valores de los parámetros del modelo que minimicen la función de costo
- 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 por la escalabilidad en cuanto al tamaño del dataset y del modelo





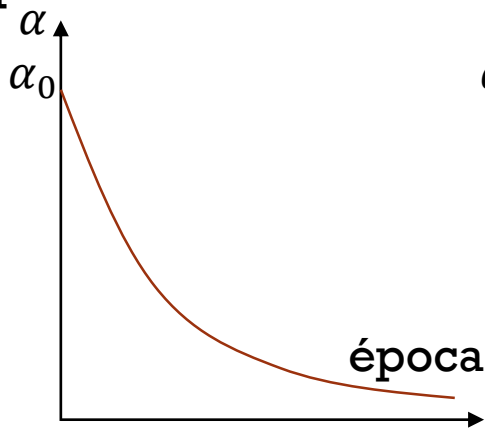
# TALLER: MINI-BATCH CON MNIST

Modificar el taller de gradient descent de MNIST agregándole un ciclo interno que se encarga del procesamiento de mini-batches.



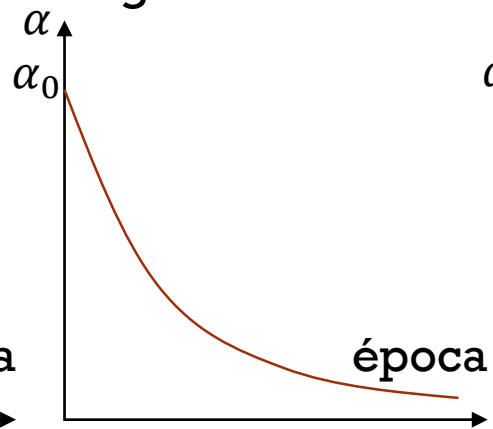
# OPTIMIZADORES

- **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



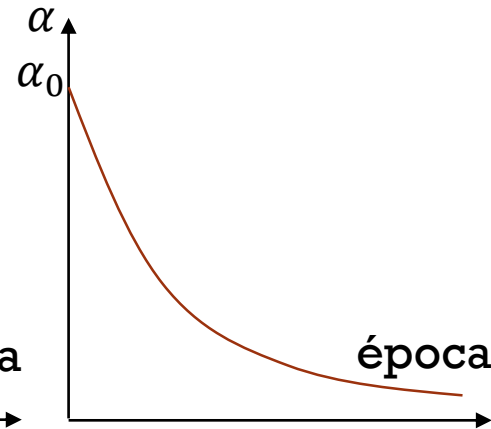
1/t Decay:

$$\alpha := \frac{\alpha_0}{1 + \text{decay} * \text{epoca}}$$

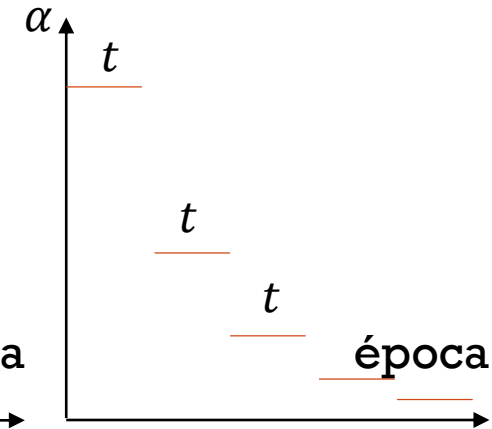


Exponential decay:

$$\alpha := \beta^{\text{epoca}}$$



$$\alpha := \frac{k}{\sqrt{\text{epoca}}} * \alpha_0$$



Step decay

$$\alpha(t) := \frac{1}{k} * \alpha(t - 1)$$



# 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
- Cada parámetro tiene entonces una tasa de aprendizaje diferente, basada en las actualizaciones anteriores del mismo. El problema es que siempre es decreciente por la acumulación sucesiva sin control de los gradientes al cuadrado → **AdaDelta**
- Se aplica en problemas con datos dispersos (tratamiento de textos, sistemas de recomendación)

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

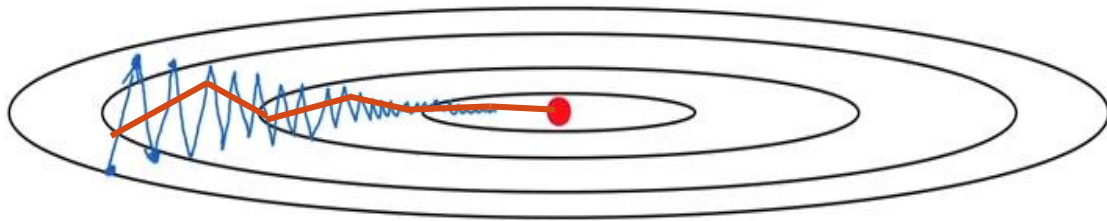
# OPTIMIZADORES

## ▪ Momentum:

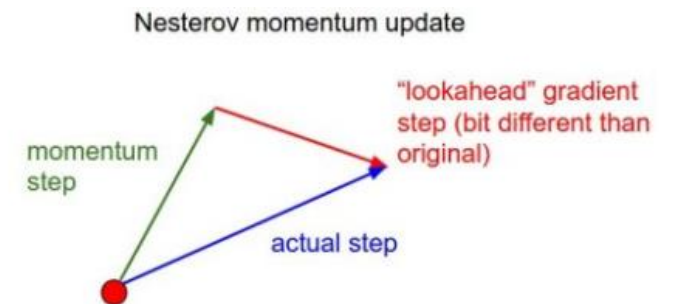
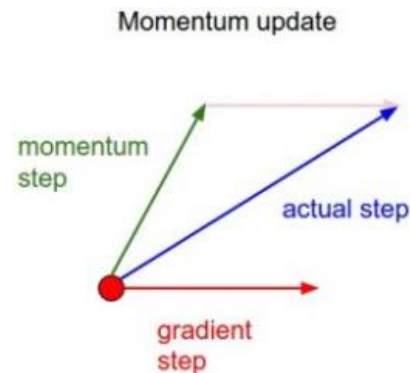
- Puede que el gradiente con respecto a uno de los parámetros sea más importante que con respecto a otros y contrario a la dirección de mayor minimización del costo
- la idea es utilizar no solo el gradiente del paso actual, sino los de los últimos pasos, haciendo un promedio ponderado exponencial (usualmente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

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

- Nesterov: promediar con el gradiente del momento



Andrew Ng, Coursera, 2017



<http://cs231n.github.io/neural-networks-3/>





# 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. Combina momentum y RMSProp (los 2 momentos):

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

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

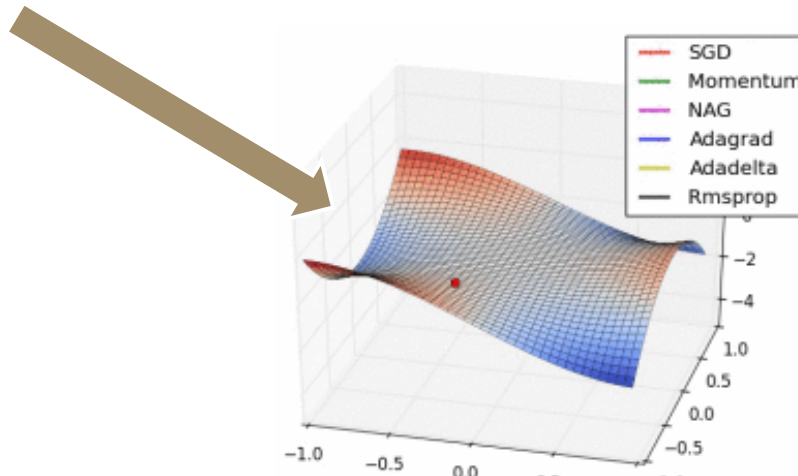
$$\theta_i := \theta_i - \alpha * \text{Momentum}(\theta_i) * \frac{1}{\sqrt{S(\theta_i) + \epsilon}}$$

- 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

# AGENDA



**Deep Learning**



**Optimizadores**



# FRAMEWORKS DE DL



TensorFlow



theano






# TENSORFLOW




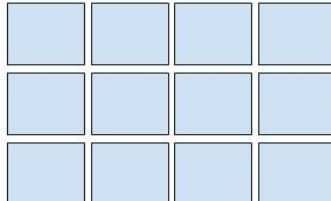
# TENSORES

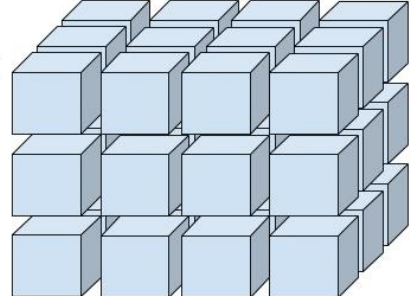
- Los datos se representan en forma de tensores: arrays multi dimensionales
- El **rango** de un tensor es el número de ejes que este tiene, cada rango tiene un número de dimensiones determinado
- Por convención,
  - los **escalares** son tensores de rango 0
  - el **primer eje** siempre será utilizado 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 largo
    - **channels-last**: primer eje para instancias, segundo para y tercero para alto y largo, cuarto para los canales de color.
  - Los **videos** siguen las mismas dos convenciones de las imágenes, teniendo en cuenta que cada frame es una imagen. En el caso de **channels-last**, se tendrían entonces los ejes arreglados de la siguiente manera: instancias, frames, alto, largo, color.



# TENSORFLOW



- Python Framework, liberado por Google en noviembre del 2015. Librería de alto nivel para computación numérica a partir de grafos de flujos de datos (**dataflow graphs**), para implementar y desplegar sistemas de Deep Learning o ML en general
- Los cálculos se hacen en implementaciones nativas (C++), aunque el código es **portable** a diferentes sistemas (HW, distribuido, cloud) sin necesidad de adaptaciones
- Flexibilidad: no es necesario encargarse de detalles de bajo nivel (e.g. la codificación de los gradientes)
- Interfaz con GPUs (CUDA) para procesamiento paralelo, sin necesidad de ocuparse de NINGUN detalle de programación de bajo nivel
- Modelos disponibles (Transfer Learning)

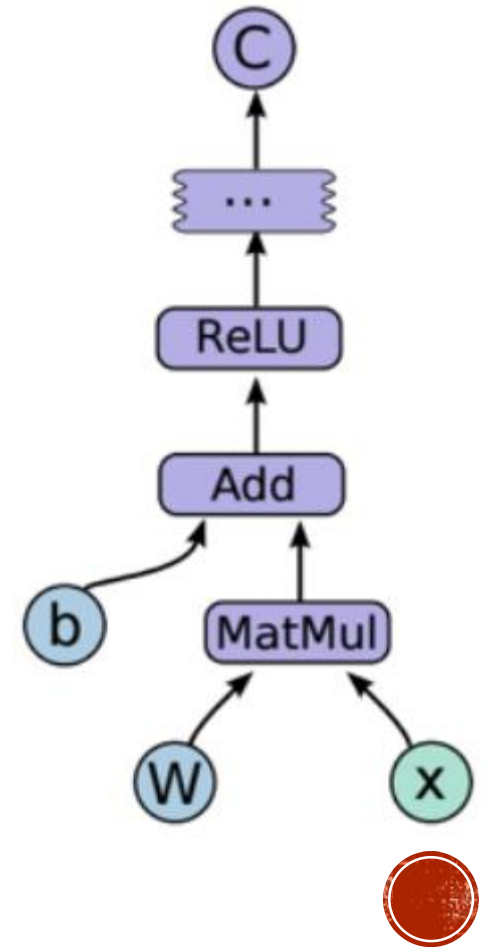




# TENSORFLOW



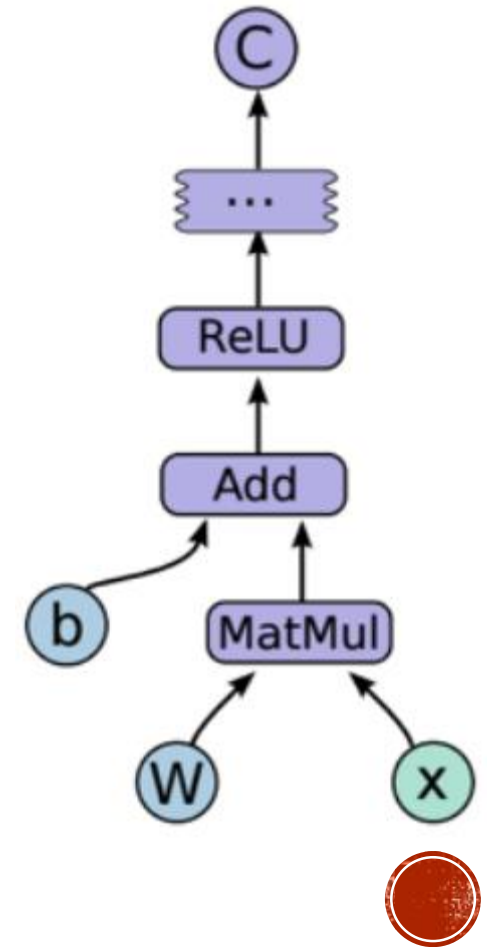
- Librerías de alto nivel construidas sobre TensorFlow: **Keras**, TF-Slim, TFLearn.
- **TensorBoard** permite el monitoreo, debug y visualización de los procesos de entrenamiento
- Para ahorrar los costos del paso de Python a las implementaciones nativas (lazy evaluation), se trata de definir todo el esquema de operaciones en un **dataflow graph**:
  - Los nodos son operaciones (adiciones, multiplicaciones)
  - Los arcos son datos (en forma de tensores)
  - Los cálculos de los gradientes se simplifican (automatización, chain rule)
  - Cada operación se puede ver como una función que se puede evaluar en cualquier momento



# TENSORFLOW



- **Constantes:** (`tf.constant`) nodos con valores estáticos.
- **Variables:** (`tf.Variable`) nodos con valores que pueden ser modificados por TF al realizar operaciones ( $\rightarrow$  los parámetros a optimizar). Sus valores se pueden guardar en disco en cualquier momento.
- **Placeholders:** (`tf.placeholder`) nodos cuyos valores son alimentados desde afuera de TF ( $\rightarrow$  los datasets para aprender o evaluar). Inicialmente no tienen valores asignados; se les especifica un tipo de dato y un shape para que el grafo sepa lo que debe calcular.
- **Operaciones matemáticas:** (e.g. `tf.matmul`, `tf.add`, `tf.relu`) nodos que se alimentan de otros nodos par realizar operaciones matemáticas. Es importante aclarar que son operaciones de TF, no de NumPy.
- **Operaciones matemáticas nativas a Python:** TF convierte automáticamente sumas, restas y multiplicación element-wise a operaciones (y nodos) de TF.
- Cada nodo en TF tiene asociada la función de gradiente correspondiente, que se back-propaga a través del grafo usando chain rule, automáticamente.
- Abajo nivel, TF define mas nodos de los que se declaran programáticamente



# TENSORFLOW



- A partir de **name scopes**, es posible manejar un espacio de nombres para los nodos creados (operaciones). Es de mucha utilidad en el caso de grafos muy complejos, permitiendo organizarlo mejor
- Una **sesión** esta relacionada a un contexto de ejecución (i.e una CPU, una GPU, por defecto utiliza una CPU) y soporta la ejecución de un dataflow graph. La definición de la estructura del grafo no implica ningún cálculo hasta que este no se realiza a través de una sesión.
- Google está desarrollando su propio circuito TPU: Tensor Processing Unit
- Una sesión se ejecuta recibiendo como parámetros:
  - **Fetches**: lista de nodos del grafo que se quiere ejecutar. Estos serán retornados como salidas de la ejecución
  - **Feeds**: diccionario de valores mapeando nodos de entrada (placeholders) del grafo a los valores que se les asignarán antes de la ejecución.



# TENSORFLOW

- Se puede definir y ejecutar nodos para:
  - Inicializar las variables
  - Funciones de costo
  - Funciones de métricas
  - Funciones de predicción que utilicen el modelo (grafo) previamente entrenado
  - Funciones de optimización (e.g. un paso de gradient descent que toma como entrada la función de costo)
  - Obtener valores intermedarios

TensorFlow operator	Shortcut	Description
<code>tf.add()</code>	<code>a + b</code>	Adds a and b, element-wise.
<code>tf.multiply()</code>	<code>a * b</code>	Multiplies a and b, element-wise.
<code>tf.subtract()</code>	<code>a - b</code>	Subtracts a from b, element-wise.
<code>tf.divide()</code>	<code>a / b</code>	Computes Python-style division of a by b.
<code>tf.pow()</code>	<code>a ** b</code>	Returns the result of raising each element in a to its corresponding element b, element-wise.
<code>tf.mod()</code>	<code>a % b</code>	Returns the element-wise modulo.
<code>tf.logical_and()</code>	<code>a &amp; b</code>	Returns the truth table of a & b, element-wise. dtype must be <code>tf.bool</code> .
<code>tf.greater()</code>	<code>a &gt; b</code>	Returns the truth table of a > b, element-wise.
<code>tf.greater_equal()</code>	<code>a &gt;= b</code>	Returns the truth table of a >= b, element-wise.
<code>tf.less_equal()</code>	<code>a &lt;= b</code>	Returns the truth table of a <= b, element-wise.
<code>tf.less()</code>	<code>a &lt; b</code>	Returns the truth table of a < b, element-wise.
<code>tf.negative()</code>	<code>-a</code>	Returns the negative value of each element in a.
<code>tf.logical_not()</code>	<code>~a</code>	Returns the logical NOT of each element in a. Only compatible with Tensor objects with dtype of <code>tf.bool</code> .
<code>tf.abs()</code>	<code>abs(a)</code>	Returns the absolute value of each element in a.
<code>tf.logical_or()</code>	<code>a   b</code>	Returns the truth table of a   b, element-wise. dtype must be <code>tf.bool</code> .





# TENSORFLOW



- **Optimizadores:** En el paquete `tf.train`, TF consta de optimizadores que controlan el aprendizaje de los parámetros de las redes a través de un proceso de descenso de gradiente de manera automática dadas las funciones de activación disponibles: `GradientDescentOptimizer`, `AdamOptimizer`, `RMSPropOptimizer`, ...
- **Módulo Layers:** Se trata de una librería que abstrae la definición de las variables intermedias necesarias para la creación de las diferentes capas de una red neuronal en TensorFlow. Por ejemplo, `tf.layers.dense` es una capa que conecta todos los inputs con todas las neuronas de manera directa tradicional.



# TENSORFLOW



**Persistencia de modelos en TF:** una vez un modelo está entrenado, se puede persistir su estado (el valor de los parámetros) guardándolo en un archivo

- Se puede crear un nodo `tf.train.Saver`, y llamar a su método `save()` enviando como parámetro la sesión (donde se ejecuta el modelo) y el nombre del archivo en el disco duro, creando 3 archivos de salida (con un solo nombre y 3 extensiones *meta*, *data* e *index*)
- Se puede luego cargar el modelo en memoria, lo que requiere:
  - Reconstruir el mismo grafo original con los mismos nodos y nombres
  - Cargar el archivo *meta* utilizando `tf.train.import_meta_graph`.
  - Restaurar los parámetros en la sesión actual utilizando `saver.restore(sesión, "archivo sin extensión")`
- Para poder pasar de un tensor a un numpy array, se puede utilizar `tf.reshape`.



# TALLER INTRO A TENSORFLOW

Entender y ejecutar el taller introductorio de TensorFlow.



# TALLER MNIST CON TENSORFLOW

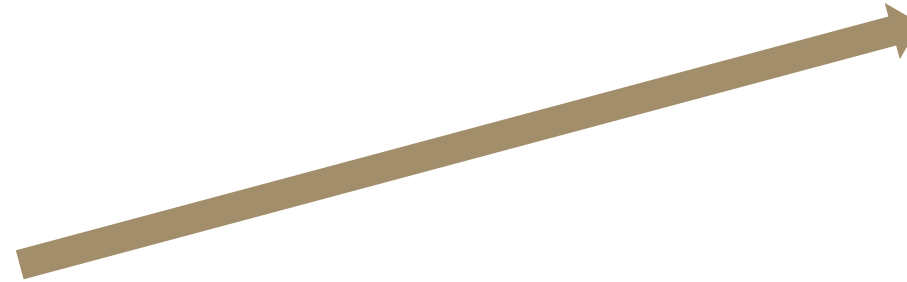
Desarrollar un clasificador de MNIST a partir de la librería TensorFlow y de su módulo tf.layers.



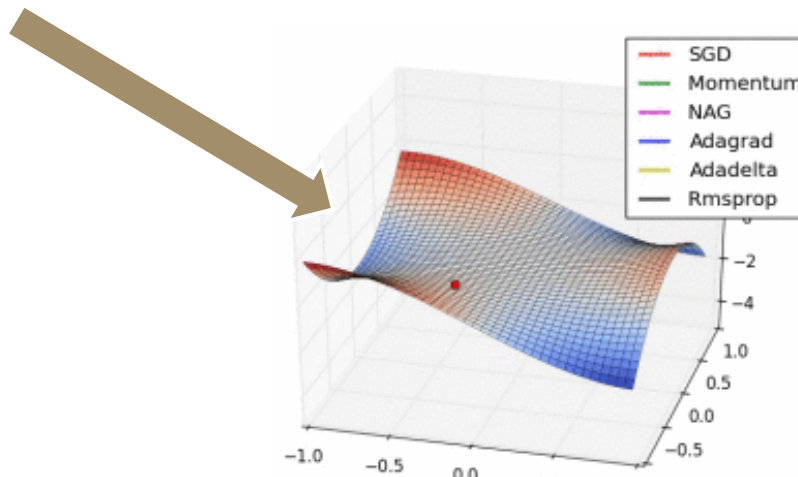
# AGENDA



**Deep Learning**



**Keras**



**Optimizadores**





# KERAS

- El framework más amigable e intuitivo. Agrega un nivel de abstracción de más trabajando por encima de otros frameworks que usa como **back-end** (TensorFlow Theano, o Microsoft CNTK). Se puede acceder al back-end desde Keras.
- Desde 2017 hace parte de **TensorFlow** como una interface de alto nivel.
- Incluye implementaciones comunes de capas, optimizadores, funciones de activación, funciones objetivo, y métodos para pre-tratamiento de imágenes y texto
- Modelos en Keras pueden ser utilizados en Android o IOS, en la web o en la JVM
- Keras tiene su propio grafo computacional que se comunica con el back-end (en nuestro caso con el grafo computacional de TF)
- Tiene dos tipos de crear modelos (organizaciones de capas), a través de dos APIs diferentes: **Sequential** y **Functional**.



# KERAS

- **Sequential:** modelos sencillos, con capas posicionadas una después de otra, donde cada capa solo recibe inputs de la capa anterior y envía resultados a la siguiente.
  - Se crea el modelo primero. Se agregan capas una encima de la otra
  - El modelo necesita saber el tipo de input que recibirá (un shape que no incluye el batch necesariamente)

```
from keras.models import Sequential
from keras.layers import Dense, Activation

model = Sequential([
    Dense(32, input_shape=(784,)),
    Activation('relu'),
    Dense(10),
    Activation('softmax'),
])

model = Sequential()
model.add(Dense(32, input_dim=784))
model.add(Activation('relu'))
```

[keras.io/getting-started/sequential-model-guide/](https://keras.io/getting-started/sequential-model-guide/)



# KERAS

- **Functional:** permite mucha mas flexibilidad, con estructuras diversas, conexiones cíclicas (recursivas), múltiples inputs y outputs, reutilización de capas por diferentes modelos
  - Cada capa se crea especificando el(los) tensor(es) que recibe como entrada(s), y produce tensor como salida al ser ejecutada
  - Un modelo se define al especificar tensores de entrada y de salida, y puede ser luego utilizado como una capa más de otros modelos
  - Un modelo puede tener varias salidas, cada una con su función de perdida; al compilarlo se debe definir su importancia relativa (pesos)

```
from keras.layers import Input, Dense
from keras.models import Model

# This returns a tensor
inputs = Input(shape=(784,))

# a layer instance is callable on a tensor, and returns a tensor
x = Dense(64, activation='relu')(inputs)
x = Dense(64, activation='relu')(x)
predictions = Dense(10, activation='softmax')(x)

# This creates a model that includes
# the Input layer and three Dense layers
model = Model(inputs=inputs, outputs=predictions)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(data, labels) # starts training
```

[keras.io/getting-started/functional-api-guide/](https://keras.io/getting-started/functional-api-guide/)



# KERAS

Independientemente del API utilizado, después de ser definidos, los modelos deben pasar por un proceso de compilación, entrenamiento, y eventual uso (predicción):

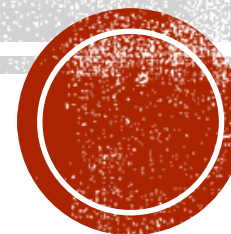
- **Compilación:** los modelos deben ser compilados, para lo que se debe definir un **optimizador** (pre-creado o instanciado con parámetros especificados), una **función de pérdida** a minimizar y las **métricas** que se desea monitorear durante el entrenamiento
- **Entrenamiento:** se debe llamar un método de **fit** del modelo, pasando los tensores de input y de salida esperada en forma de numpy arrays, y opcionalmente el número de **épocas**, el **batch\_size**, el porcentaje de los datos que se va a acordar a **validación** que se quiere utilizar para hacerle seguimiento a las métricas, entre otros.
- **Predicción:** los modelos se usan sobre un set de evaluación o test a través de un método **predict**.

# TALLER MNIST CON KERAS

Desarrollar un clasificador de MNIST a partir de la librería Keras, utilizando los frameworks secuencial y funcional.



# GRACIAS





# REFERENCIAS

- *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
- *Python Machine Learning (2nd ed.)*, Sebastian Raschka & Vahid Mirjalili, Packt, 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

