



MULTIPERCEPTRON

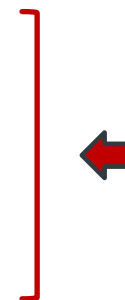
TENSORFLOW
KERAS



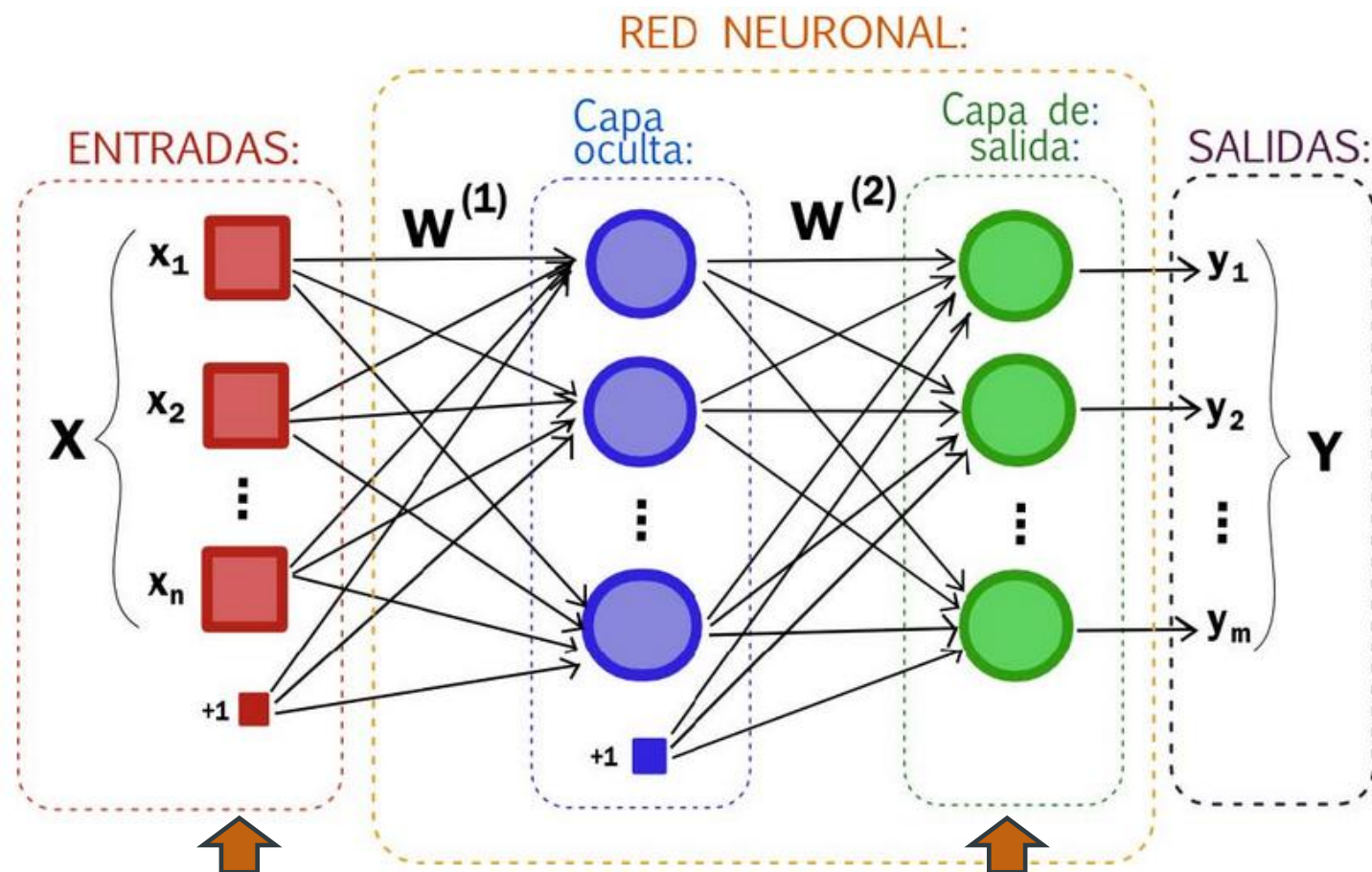
RESUMEN

Resolución de una tarea de clasificación

- Conjunto de datos etiquetados (aprendizaje supervisado)
- Definición de la arquitectura de la red
 - Número de capas y tamaño de cada una
 - Función de activación a usar en cada capa
- Entrenamiento
 - Función de error
 - Técnica de optimización para reducir el error
- Evaluar el modelo

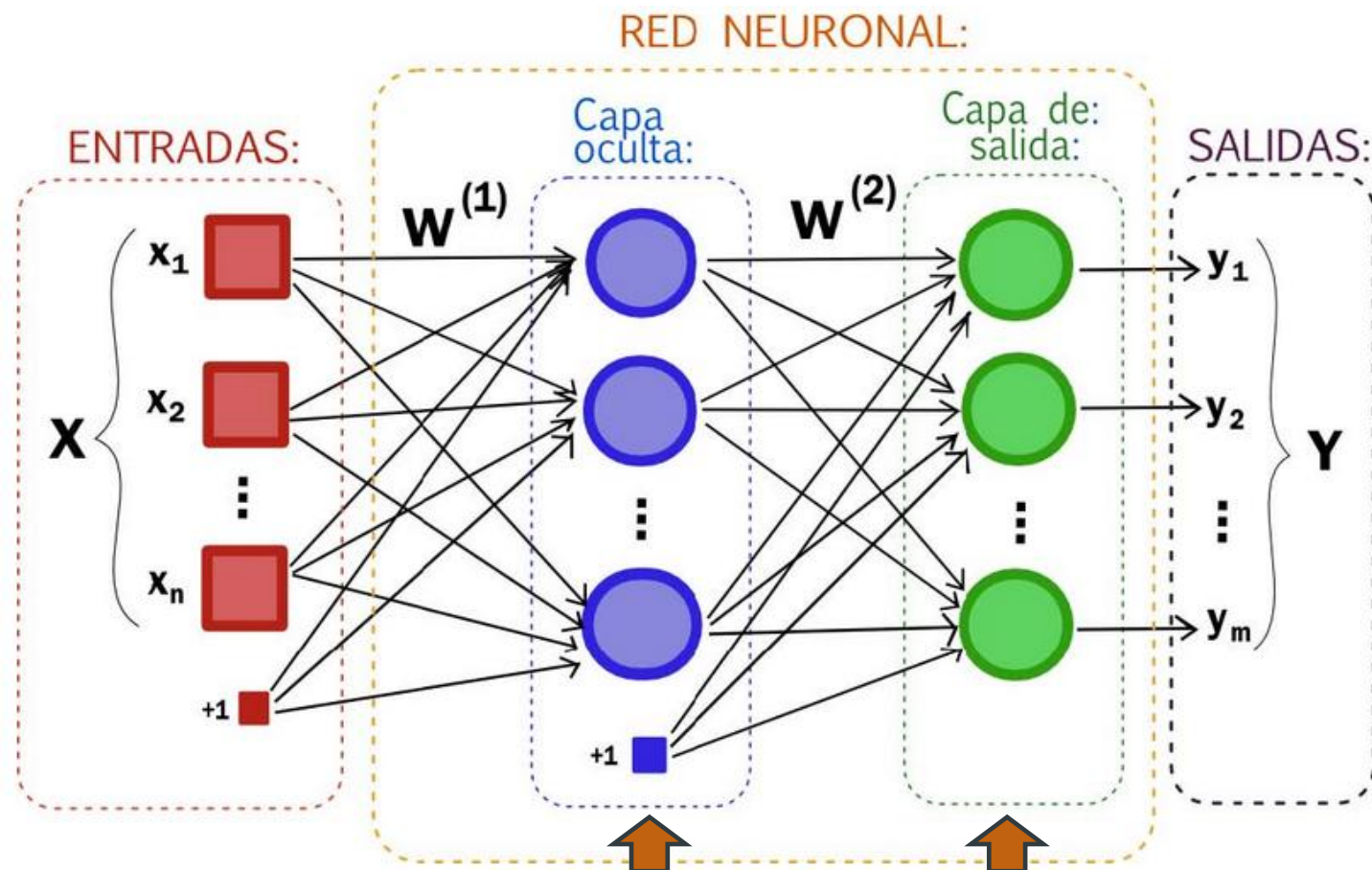


ARQUITECTURA DE LA RED



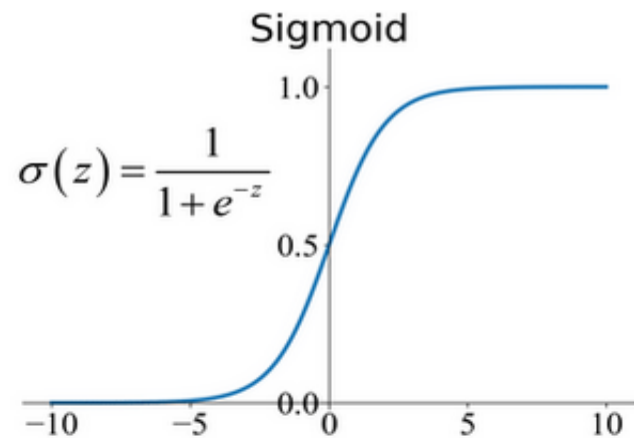
Las dimensiones de las capas de entrada y salida las define el problema

ARQUITECTURA DE LA RED

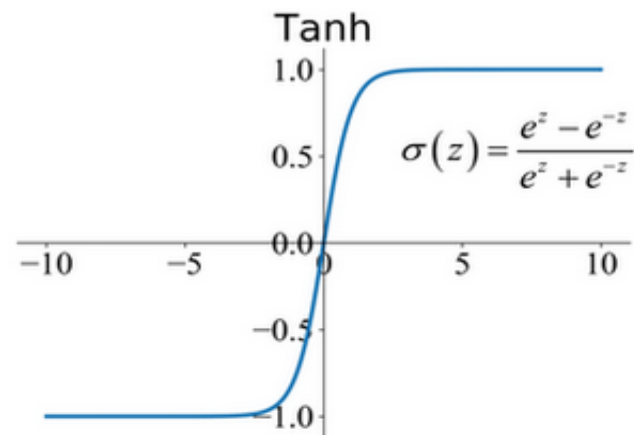
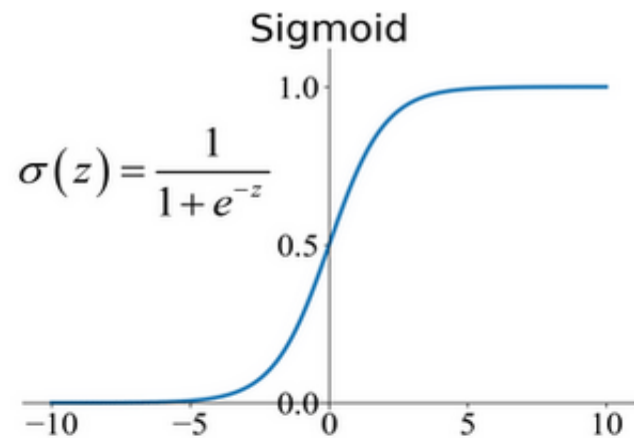


Su respuesta depende de la **Función de activación** elegida

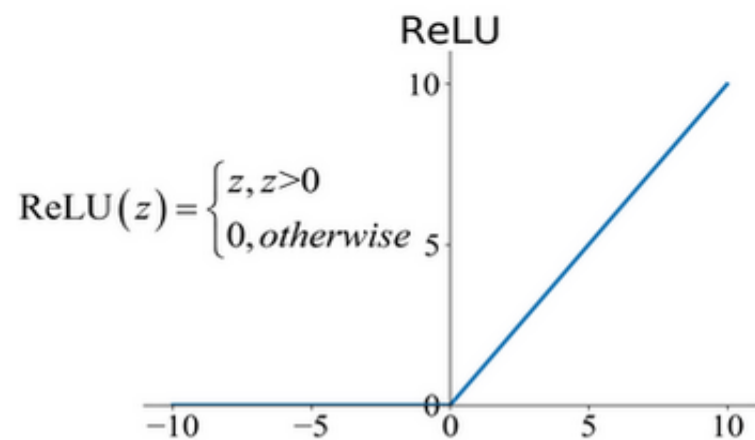
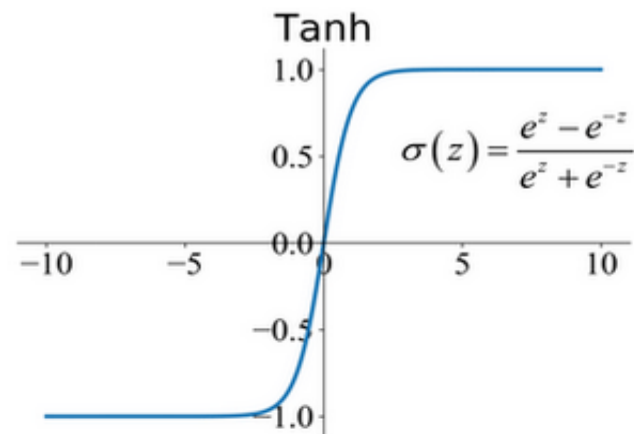
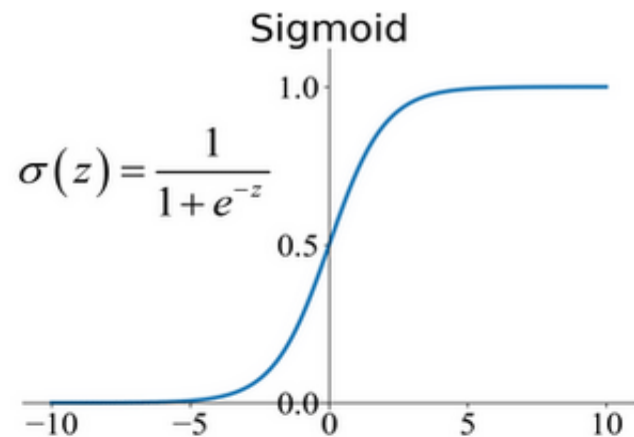
FUNCIONES DE ACTIVACIÓN



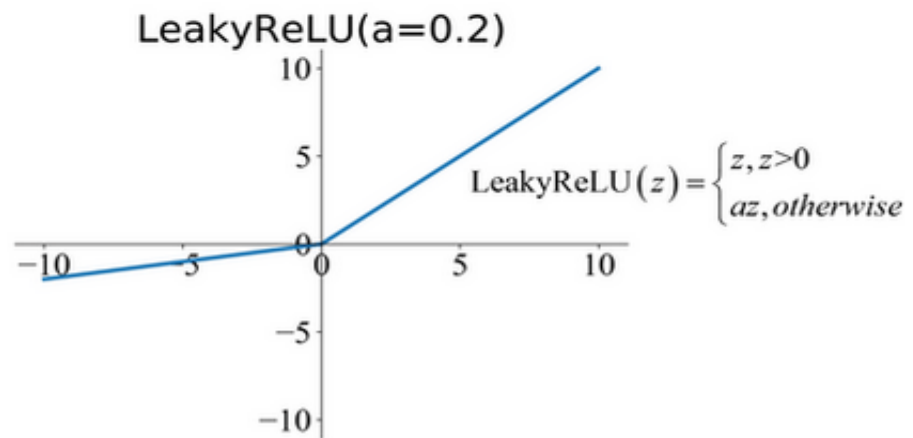
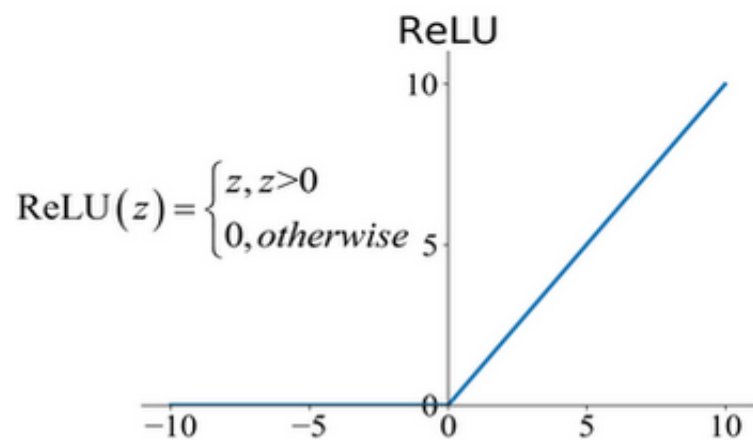
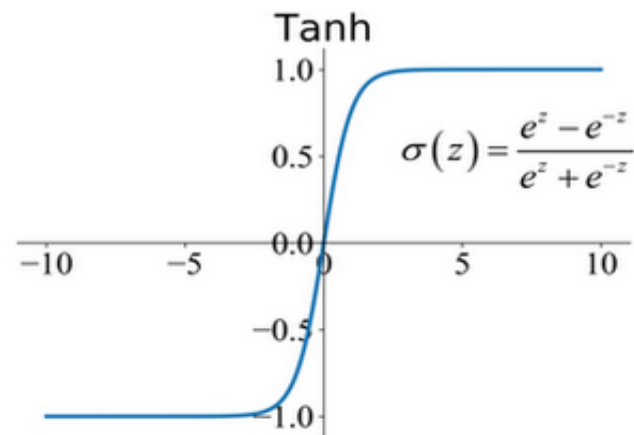
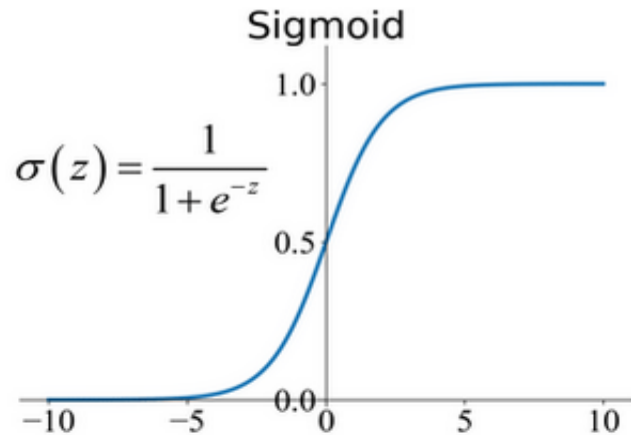
FUNCIONES DE ACTIVACIÓN



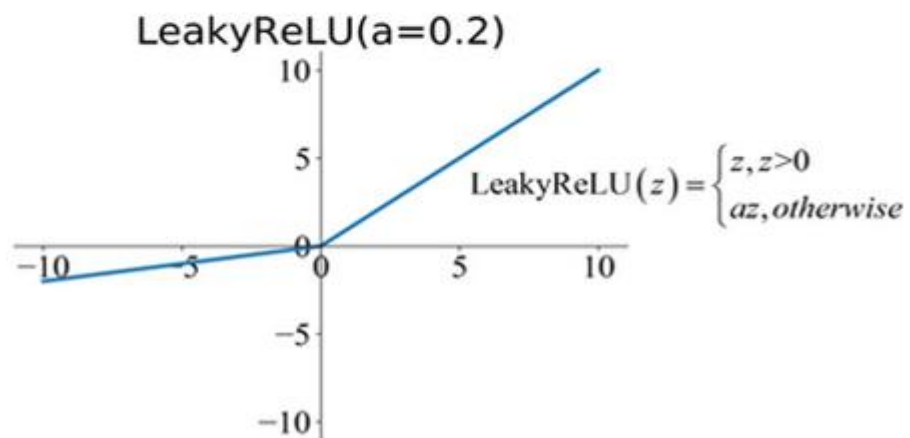
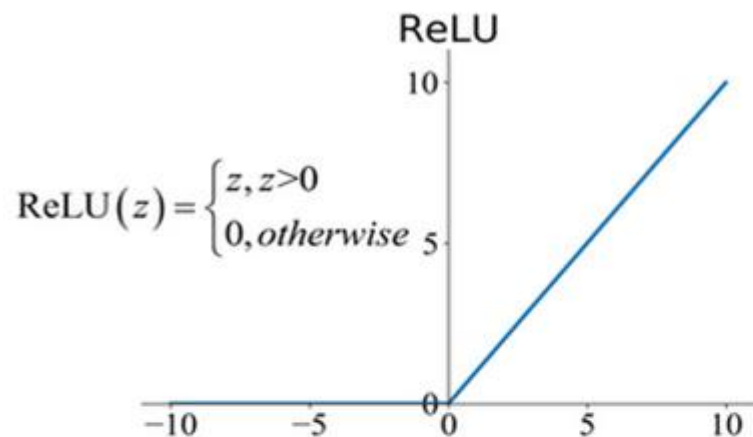
FUNCIONES DE ACTIVACIÓN



FUNCIONES DE ACTIVACIÓN



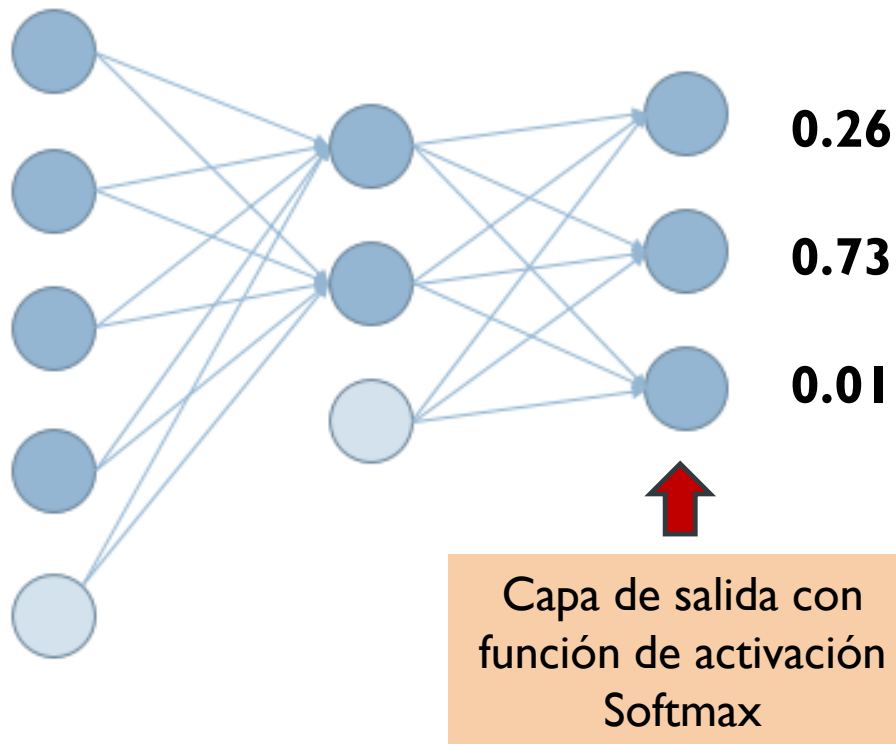
RELU (UNIDAD LINEAL RECTIFICADA)



- Velocidad de aprendizaje (derivada)
- Velocidad de cómputo (fácil de calcular)
- Activa sólo algunas neuronas

FUNCIÓN SOFTMAX

- Se utiliza como función de activación en la última capa para normalizar la salida de la red de manera que los valores sumen 1.



$$neta_j = \sum_i w_{ji} x_i + b_j$$

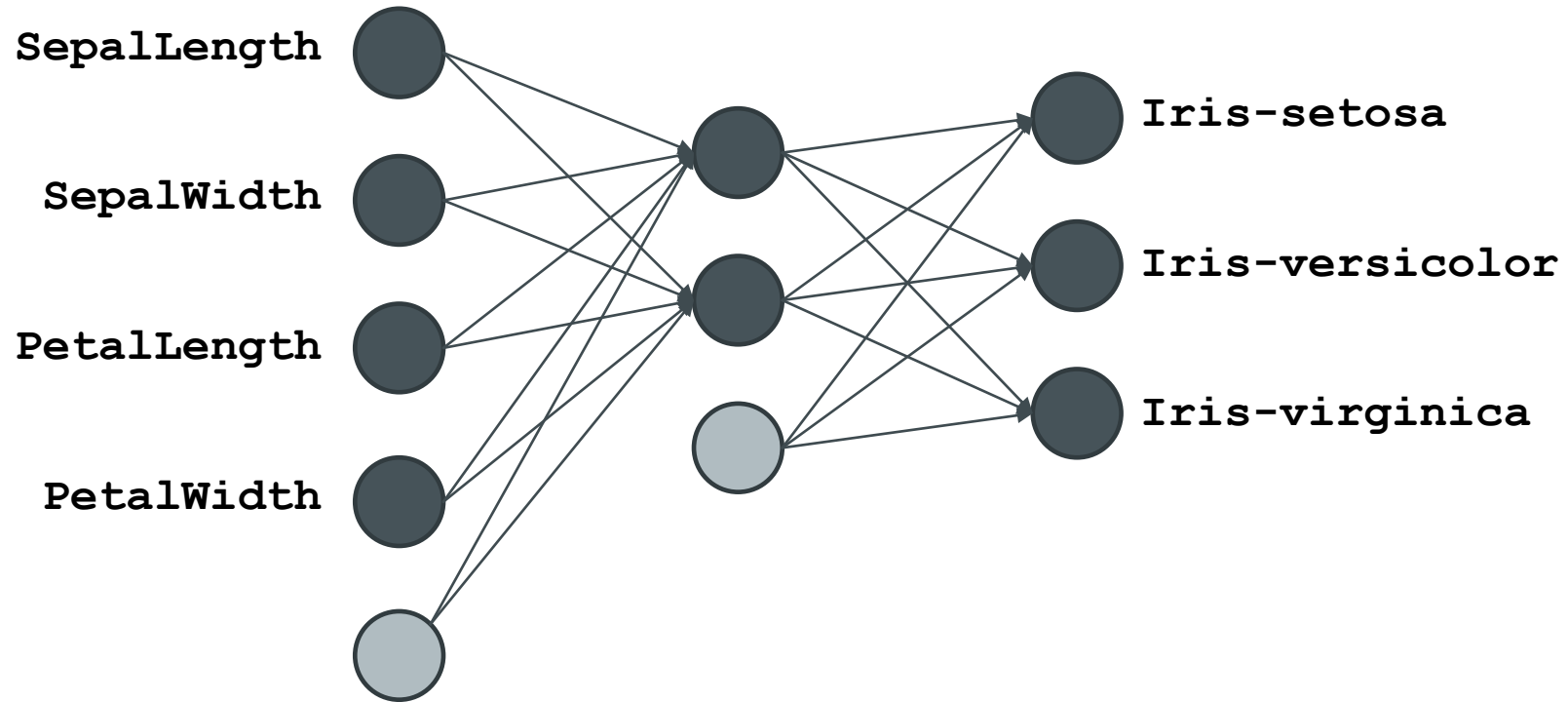
$$y_j = \frac{e^{neta_j}}{\sum_k e^{neta_k}}$$

EJEMPLO: CLASIFICACIÓN DE FLORES DE IRIS

Id	sepalength	sepalwidth	petallength	petalwidth	class
1	5,1	3,5	1,4	0,2	Iris-setosa
2	4,9	3,0	1,4	0,2	Iris-setosa
...
95	5,6	2,7	4,2	1,3	Iris-versicolor
96	5,7	3,0	4,2	1,2	Iris-versicolor
97	5,7	2,9	4,2	1,3	Iris-versicolor
...
149	6,2	3,4	5,4	2,3	Iris-virginica
150	5,9	3,0	5,1	1,8	Iris-virginica

<https://archive.ics.uci.edu/ml/datasets/Iris>

EJEMPLO: CLASIFICACIÓN DE FLORES DE IRIS



KERAS

- **Keras** es una biblioteca de código abierto escrita en Python que facilita la creación de modelos complejos de aprendizaje profundo
- Características
 - ▣ Prototipado rápido del modelo.
 - ▣ De alto nivel (programación a nivel de capa)
 - ▣ Actualmente integrada con **Tensorflow**, una plataforma de código abierto para el desarrollo y la implementación de modelos de aprendizaje automático.

veamos cómo usar Keras dentro de Tensorflow

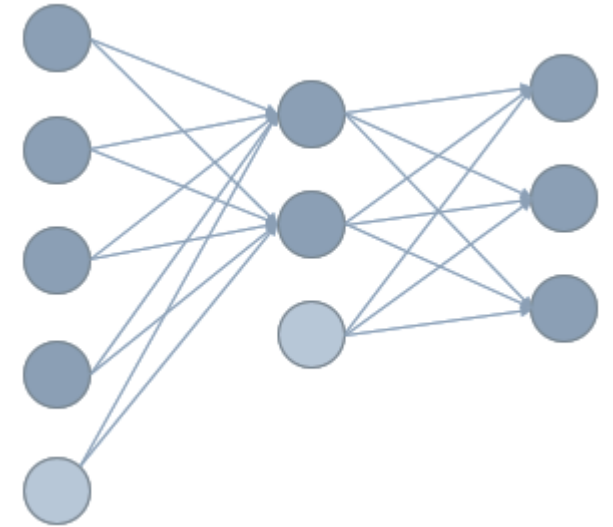


CONSTRUCCIÓN DEL MODELO

```
from tensorflow.keras.models import Sequential  
from tensorflow.keras.layers import Dense, Input
```

Crear un modelo de capas secuenciales

```
model=Sequential()
```



CONSTRUCCIÓN DEL MODELO

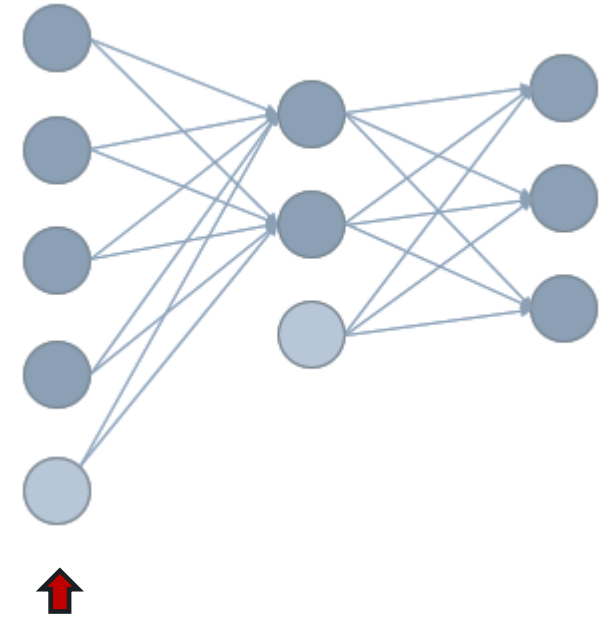
```
from tensorflow.keras.models import Sequential  
from tensorflow.keras.layers import Dense, Input
```

Crear un modelo de capas secuenciales

```
model=Sequential()
```

Agregar las capas al modelo

```
model.add(Input(shape=[4]))
```



CONSTRUCCIÓN DEL MODELO

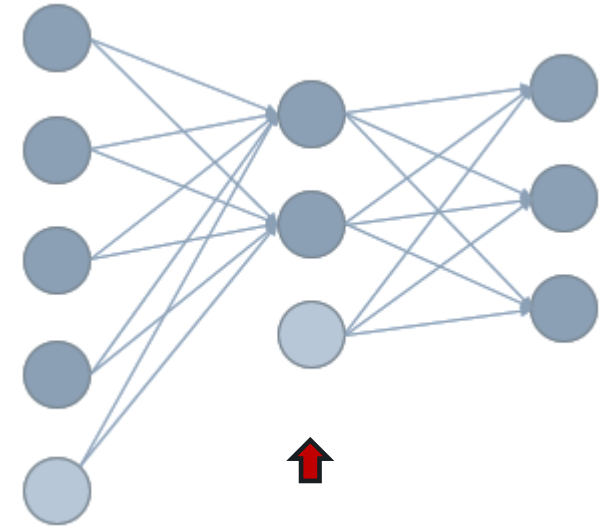
```
from tensorflow.keras.models import Sequential  
from tensorflow.keras.layers import Dense, Input
```

Crear un modelo de capas secuenciales

```
model=Sequential()
```

Agregar las capas al modelo

```
model.add(Input(shape=[4]))  
model.add(Dense(2, activation='tanh'))
```



CONSTRUCCIÓN DEL MODELO

```
from tensorflow.keras.models import Sequential  
from tensorflow.keras.layers import Dense, Input
```

Crear un modelo de capas secuenciales

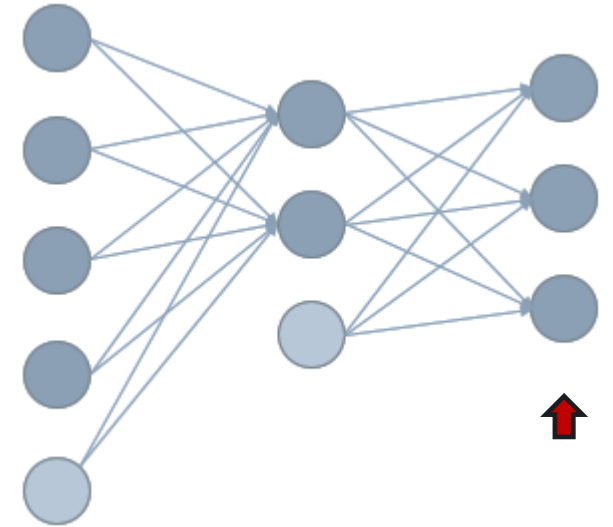
```
model=Sequential()
```

Agregar las capas al modelo

```
model.add(Input(shape=[4]))
```

```
model.add(Dense(2, activation='tanh'))
```

```
model.add(Dense(3, activation='sigmoid'))
```



CONSTRUCCIÓN DEL MODELO

```
from tensorflow.keras.models import Sequential  
from tensorflow.keras.layers import Dense, Input
```

Crear un modelo de capas secuenciales

```
model=Sequential()
```

Agregar las capas al modelo

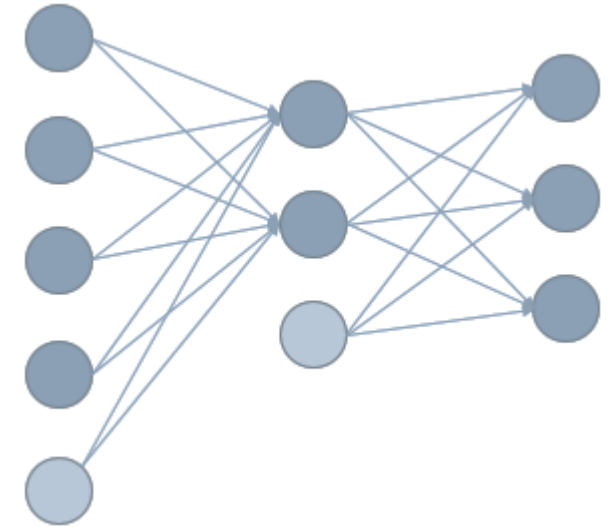
```
model.add(Input(shape=[4]))
```

```
model.add(Dense(2, activation='tanh'))
```

```
model.add(Dense(3, activation='sigmoid'))
```

Imprimir un resumen del modelo

```
model.summary()
```



Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 2)	10
dense_2 (Dense)	(None, 3)	9
Total params: 19		
Trainable params: 19		
Non-trainable params: 0		

CONFIGURACIÓN PARA ENTRENAMIENTO

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

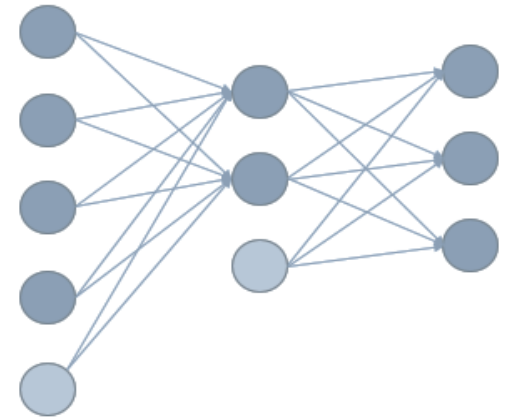
```
model=Sequential()
model.add(Dense(2, input_shape=[4], activation='tanh'))
model.add(Dense(3, activation='sigmoid'))
```

Configuración para entrenamiento

```
model.compile(optimizer='sgd', loss='mse', metrics=['accuracy'])
```

*Descenso de gradiente
estocástico*

*Error Cuadrático
Medio*



Keras_IRIS.ipynb

CONFIGURACIÓN PARA ENTRENAMIENTO

```
from keras.optimizers import SGD
```



```
from keras.models import Sequential
```

```
from keras.layers import Dense
```

```
model=Sequential()
```

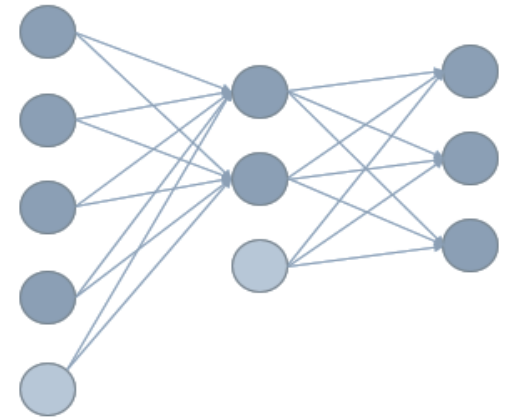
```
model.add(Dense(2, input_shape=[4], activation='tanh'))
```

```
model.add(Dense(3, activation='sigmoid'))
```

Configuración para entrenamiento

```
model.compile(optimizer=SGD(learning_rate=0.1), loss='binary_crossentropy', metrics=['accuracy'])
```

Tasa de aprendizaje



Keras_IRIS_SGD.ipynb

CONFIGURACIÓN PARA ENTRENAMIENTO

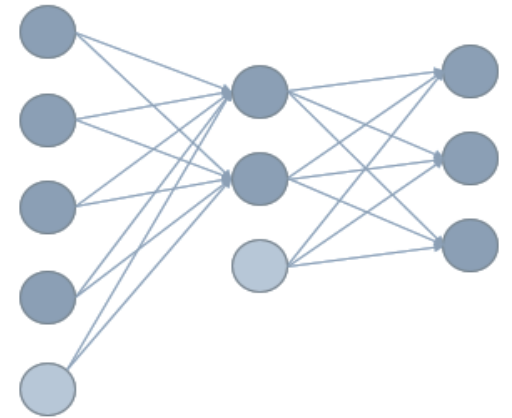
```
from keras.optimizers import SGD
from keras.models import Sequential
from keras.layers import Dense

model=Sequential()
model.add(Dense(2, input_shape=[4], activation='tanh'))
model.add(Dense(3, activation='sigmoid'))
```

Configuración para entrenamiento

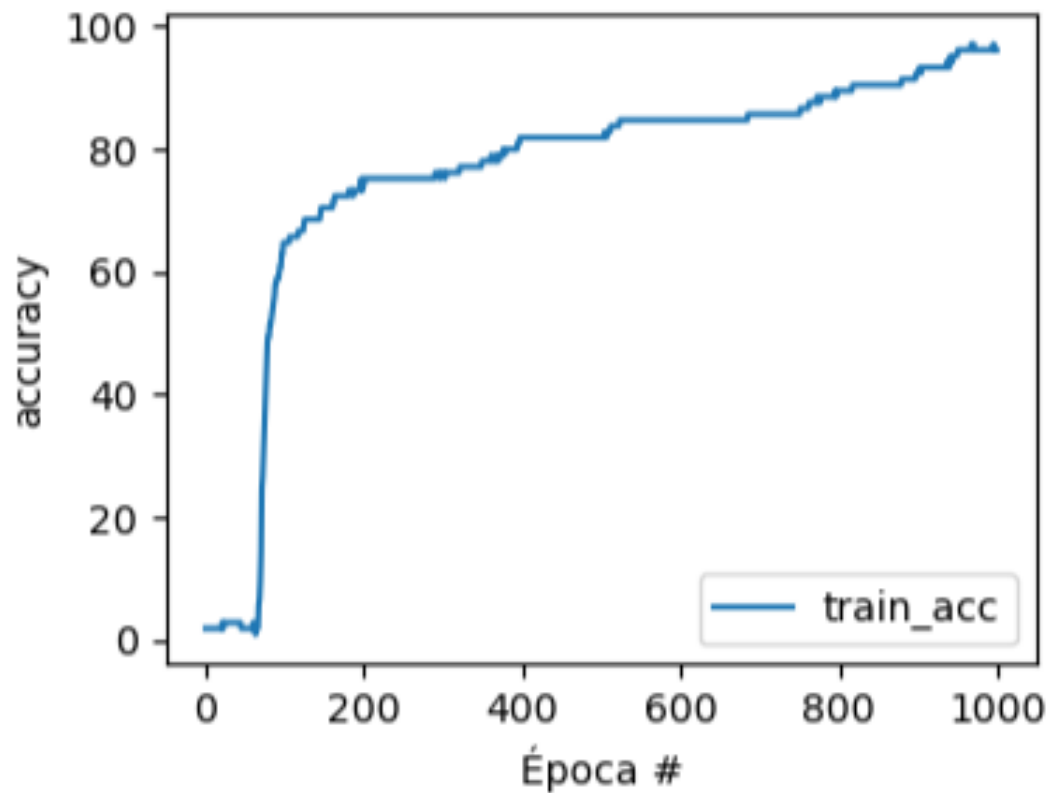
```
model.compile(optimizer=SGD(learning_rate=0.1), loss='binary_crossentropy', metrics=['accuracy'])
```

¿ debería usar **'mse'** ?

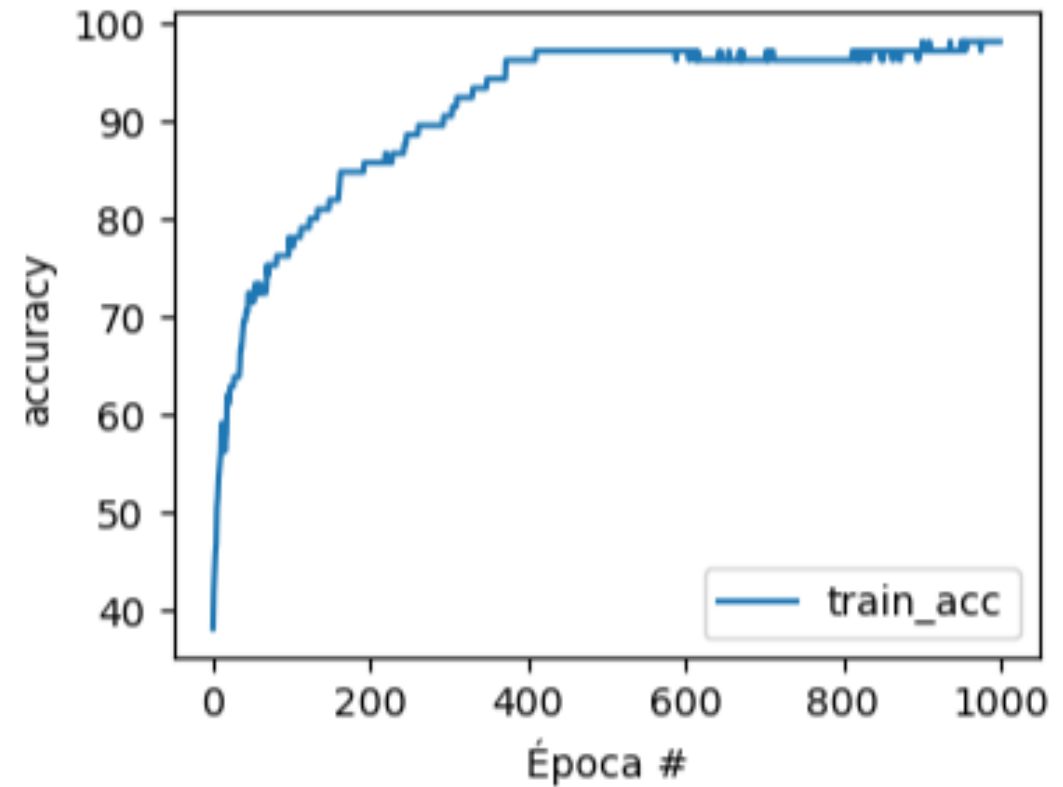


Keras_IRIS_SGD.ipynb

IRIS – FUNCIÓN DE ACTIVACIÓN ‘SIGMOID’



activation='sigmoid'
loss='mse'



activation='sigmoid'
loss='binary_crossentropy'

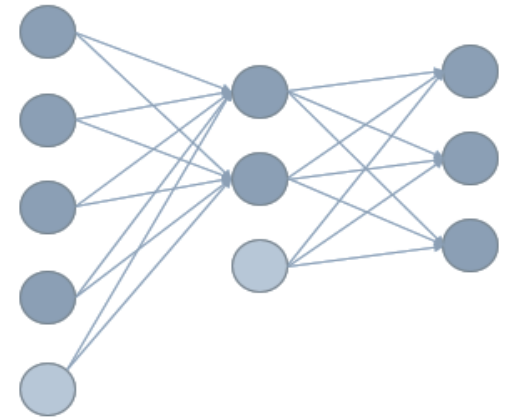
CONFIGURACIÓN PARA ENTRENAMIENTO

```
from keras.optimizers import SGD
from keras.models import Sequential
from keras.layers import Dense

model=Sequential()
model.add(Dense(2, input_shape=[4], activation='tanh'))
model.add(Dense(3, activation='softmax'))
```

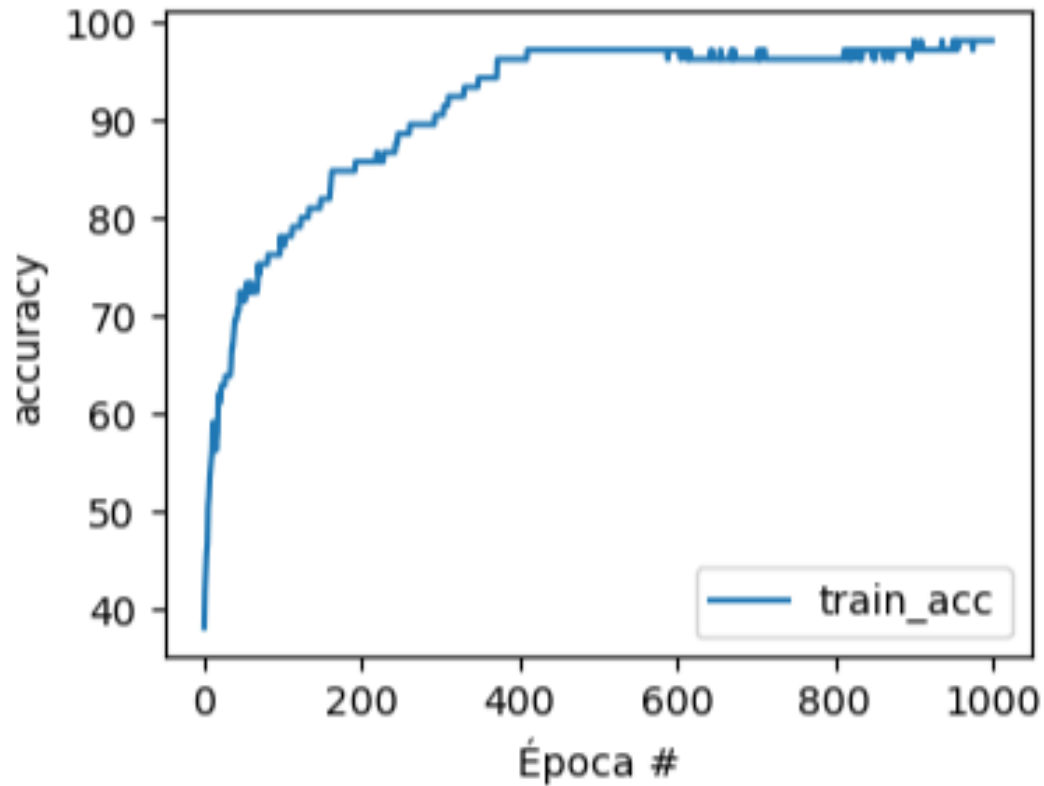
Configuración para entrenamiento

```
model.compile(optimizer=SGD(learning_rate=0.1), loss='categorical_crossentropy',
              metrics=['accuracy'])
```

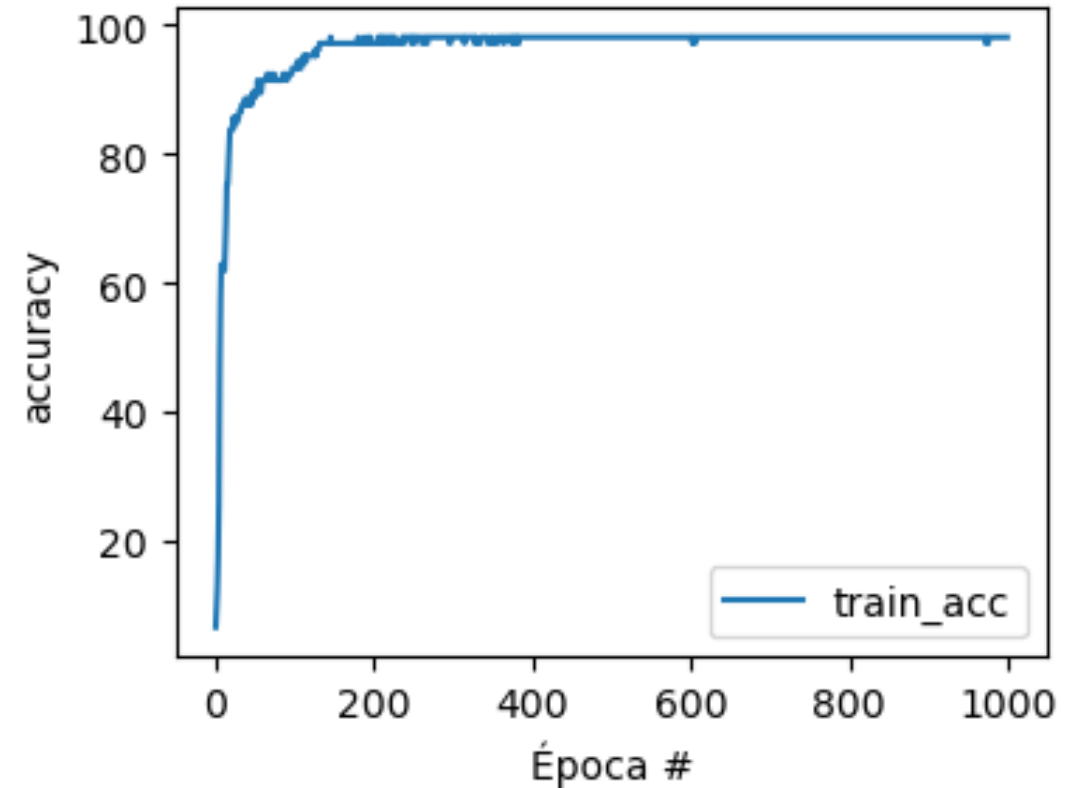


Keras_IRIS_Softmax.ipynb

IRIS – FUNCIÓN DE ACTIVACIÓN ‘SIGMOID’ vs ‘SOFTMAX’



activation='sigmoid'
loss='binary_crossentropy'



activation='softmax'
loss='categorical_crossentropy'

CONFIGURACIÓN PARA ENTRENAMIENTO

Función de activación	Función de costo
linear tanh	Error cuadrático medio
sigmoid	Error cuadrático medio Entropía cruzada binaria
softmax	Entropía cruzada categórica

CARGA DE DATOS

```
X,T = cargar_datos()
```

```
binarizer = preprocessing.LabelBinarizer()
```

```
T = binarizer.fit_transform(T)
```

X → Conjunto de ejemplos de entrada

	0	1	2	3
0	5.1	3.5	1.4	0.2
1	4.9	3	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5	3.6	1.4	0.2
5	5.4	3.9	1.7	0.4

T → Rtas esperadas para cada neurona de la capa de salida

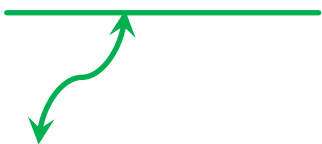
	0	1	2
0	1	0	0
1	0	1	0
2	0	0	1
3	0	0	1
4	0	1	0
5	0	0	1

ENTRENAMIENTO DEL MODELO

```
X,T = cargar_datos()    # X y T son matrices de numpy
```

```
# Entrenar el modelo
```

```
model.fit(X,T, epochs=100)
```

- 
- **Epochs:** Cantidad de veces que todo el conjunto de datos será ingresado a la red

ENTRENAMIENTO DEL MODELO

```
X,T = cargar_datos()    # X y T son matrices de numpy
```

```
# Entrenar el modelo
```

```
model.fit(X,T, epochs=100, batch_size=20)
```

- **Epochs:** Cantidad de veces que todo el conjunto de datos será ingresado a la red

- **Batch_size:** Cantidad de ejemplos que ingresan a la red con los mismos pesos.
 - A medida que ingresan los ejemplos se guardan los gradientes.
 - Al finalizar el lote, se actualizan los pesos

- Si tenemos un conjunto de datos formado por **500 ejemplos** y entrenamos el modelo durante **100 épocas** con un **tamaño de lote de 20 ejemplos**, los pesos se modificarán **25 veces** en cada época ($500/20=25$).

PREDICCIÓN DEL MODELO

```
X,T = cargar_datos()    # X y T son matrices de numpy
```

```
# Entrenar el modelo
```

```
model.fit(X,T, epochs=100, batch_size=20)
```

```
# predecir la salida del modelo
```

```
Y = model.predict(X)
```

Y tiene las mismas
dimensiones que **T**

	0	1	2
0	0.967722	0.189344	0.00421873
1	0.0372113	0.510963	0.346058
2	0.00325751	0.261545	0.917956
3	0.00823823	0.319694	0.795647
4	0.0717264	0.611822	0.171516
5	0.0134856	0.482814	0.59486

ERROR DEL MODELO

```
X,T = cargar_datos()    # X y T son matrices de numpy
```

```
# Entrenar el modelo
```

```
model.fit(X,T, epochs=100, batch_size=20)
```

```
# predecir la salida del modelo
```

```
Y = model.predict(X)
```

```
# Calcular el error del modelo
```

```
score = model.evaluate(X, T)
```

```
print('Error :', score[0])
```

```
print('Accuracy:', score[1])
```

Muestra el valor de la función de Costo y la precisión del modelo al finalizar el entrenamiento

MÉTRICAS

```
# Entrenar el modelo
```

```
model.fit ( X_train, T_train, epochs=100)
```

```
# predecir la salida del modelo
```

```
Y = model.predict(X_train)
```

```
# "invierte" la transformacion binaria
```

```
T_str = binarizer.inverse_transform(T_train)
```

```
Y_str = binarizer.inverse_transform(Y)
```

```
print("%% aciertos : %.3f" % metrics.accuracy_score(T_str, Y_str))
```

Y[:5, :]

	0	1	2
0	0.967722	0.189344	0.00421873
1	0.0372113	0.510963	0.346058
2	0.00325751	0.261545	0.917956
3	0.00823823	0.319694	0.795647
4	0.0717264	0.611822	0.171516
5	0.0134856	0.482814	0.59486

Y_str[:5]

```
['Iris-setosa' 'Iris-virginica'
 'Iris-setosa' 'Iris-virginica'
 'Iris-setosa']
```

PESOS DE LA RED

```
model.fit(...)
```

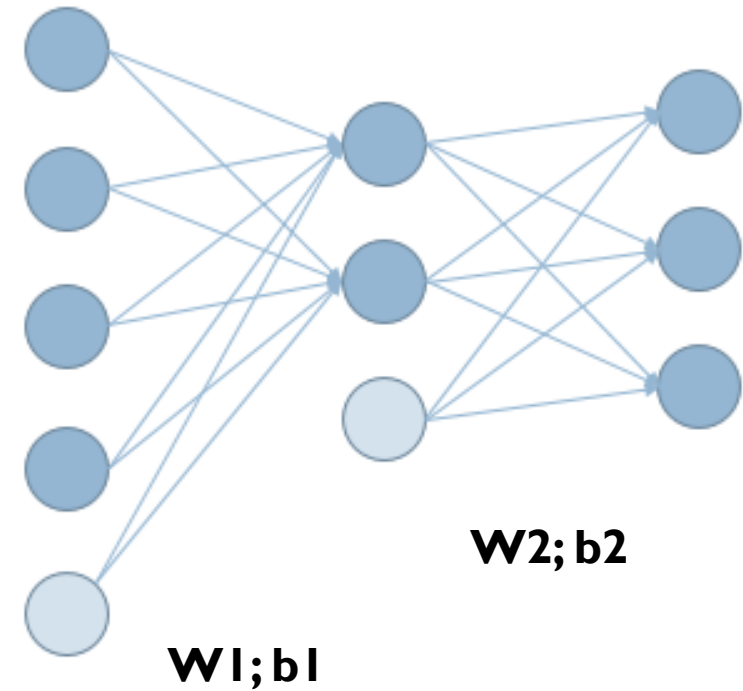
```
...
```

```
capaOculto = model.layers[0]
```

```
W1, b1 = capaOculto.get_weights()
```

```
capaSalida = model.layers[1]
```

```
W2, b2 = capaSalida.get_weights()
```



SALVAR EL MODELO

- Una vez entrenado el modelo, si los resultados han sido buenos lo guardamos para su uso posterior

OPCION 1

Guarda todo el modelo

```
model = ...  
model.fit( ... )  
...  
model.save("miModelo")
```

OPCION 2

Guarda sólo los pesos

```
model = ...  
model.fit( ... )  
...  
model.save_weights("pesos_de_miModelo")
```

Requiere definir el modelo antes de cargar

CARGAR EL MODELO

OPCION 1 – Carga el modelo completo

```
from keras.models import load_model  
model = load_model("miModelo")
```

OPCION 2 – Cargar sólo los pesos

```
model = ... (definir el modelo)  
...  
model.load_weights("pesos_de_miModelo")
```

ver
Keras_IRIS_SGD.ipynb

DEFINICIÓN DE MODELOS

- La definición de la arquitectura admite variantes:
 - Indicando la función de activación de manera separada
 - Utilizando una notación funcional
 - Definiendo capas que se combinan para formar el modelo

INDICANDO LA FUNCIÓN DE ACTIVACIÓN POR SEPARADO

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

model=Sequential()

model.add(Input(shape=[4]))

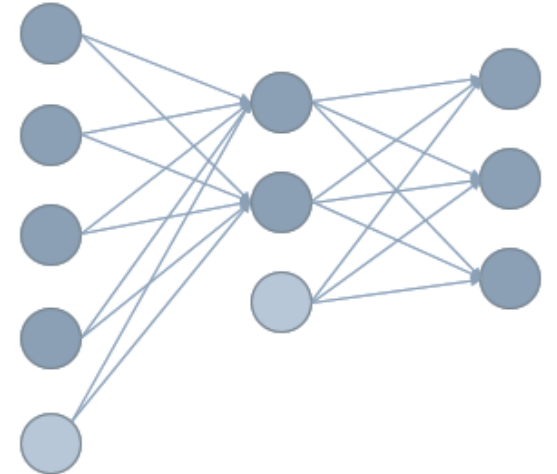
model.add(Dense(2))

model.add(Activation('tanh'))

model.add(Dense(3))

model.add(Activation('sigmoid'))

model.summary()
```



Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 2)	10
activation (Activation)	(None, 2)	0
dense_1 (Dense)	(None, 3)	9
activation_1 (Activation)	(None, 3)	0

=====
Total params: 19

Trainable params: 19

Non-trainable params: 0

USANDO UNA LISTA

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

```
model=Sequential([
    Input(shape=[4]),
    Dense(2, activation='tanh', name='Oculto'),
    Dense(3, activation='sigmoid', name='salida')])
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
=====		
Oculto (Dense)	(None, 2)	10
salida (Dense)	(None, 3)	9
=====		
Total params: 19		
Trainable params: 19		
Non-trainable params: 0		

USANDO UNA LISTA

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

model=Sequential([
    Input(shape=[4]),
    Dense(2, name='Oculto'),
    Activation('tanh', name='FunH'),
    Dense(3, name='salida'),
    Activation('sigmoid', name='FunO')])

model.summary()
```

Layer (type)	Output Shape	Param #
Oculto (Dense)	(None, 2)	10
FunH (Activation)	(None, 2)	0
salida (Dense)	(None, 3)	9
FunO (Activation)	(None, 3)	0
Total params: 19		

FUNCIONAL

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

I = Input(shape=(4,), name='Entrada')
L = Dense(units=2, activation='tanh', name='Oculto')(I)
salida=Dense(units=3, activation='sigmoid', name='salida')(L)
model = Model(inputs=I, outputs = salida)
model.summary()
```

Layer (type)	Output Shape	Param #
Entrada (InputLayer)	[(None, 4)]	0
Oculto (Dense)	(None, 2)	10
salida (Dense)	(None, 3)	9
Total params: 19		
Trainable params: 19		
Non-trainable params: 0		

FUNCIONAL

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

I = Input(shape=(4,), name='entrada')
L = Dense(units=2, name='Oculata')(I)
L = Activation('tanh', name='FunH')(L)
L = Dense(units=3, name='salida')(L)
Salida = Activation('sigmoid', name='FunO')(L)
model = Model(inputs=I, outputs = salida)
model.summary()
```

Layer (type)	Output Shape	Param #
=====		
entrada (InputLayer)	[(None, 4)]	0
Oculata (Dense)	(None, 2)	10
FunH (Activation)	(None, 2)	0
salida (Dense)	(None, 3)	9
FunO (Activation)	(None, 3)	0
=====		
Total params: 19		


DEFINIENDO CAPAS

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

I = Input(shape=(4,), name='entrada')
oculta = Dense(units=2, activation='tanh', name='Oculto')
salida = Dense(units=3, activation='sigmoid', name='salida')
red = salida(oculta(I))
model = Model(inputs=I, outputs = red)
model.summary()
```

Layer (type)	Output Shape	Param #
entrada (InputLayer)	[(None, 4)]	0
Oculto (Dense)	(None, 2)	10
salida (Dense)	(None, 3)	9
Total params: 19		

TÉCNICAS DE OPTIMIZACIÓN

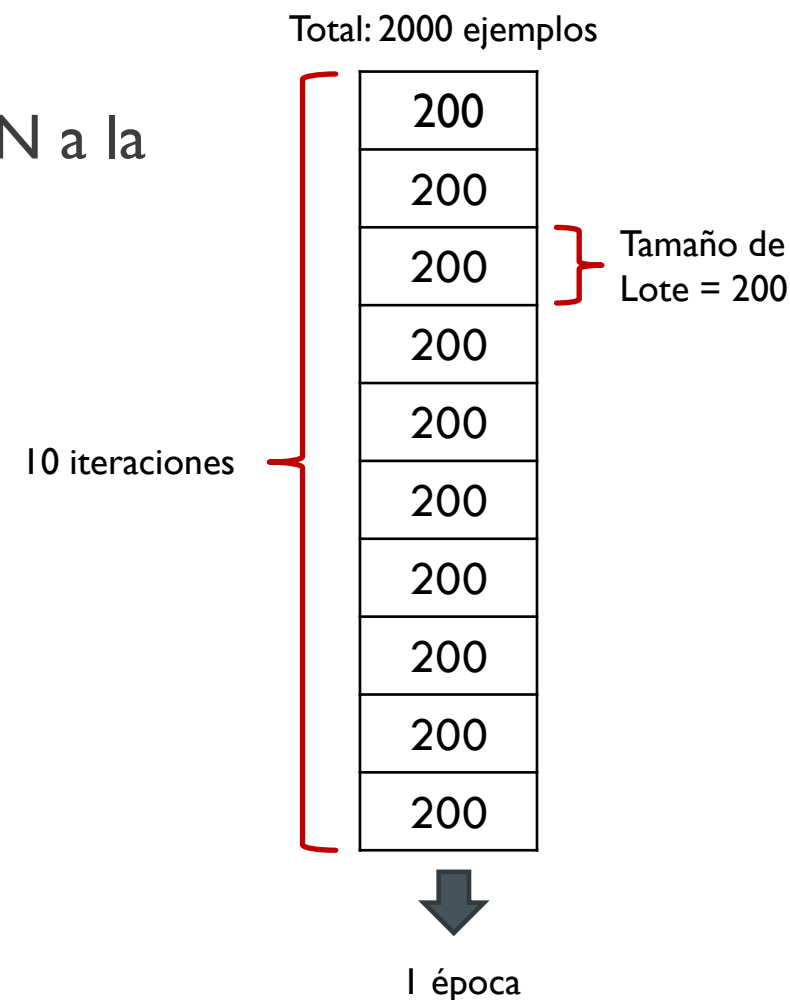
- Descenso de gradiente estocástico (SGD) y el uso de mini-lotes 
- Capacidad de generalización de la red - Sobreajuste
- Mejoras introducidas
 - Momento: utiliza información de los gradientes anteriores
 - RMSProp: considera distintas magnitudes de cambio para reducir oscilaciones
 - Adam: combina los dos anteriores. Es el más usado.

DESCENSO DE GRADIENTE EN MINI-LOTES

- En lugar de ingresar los ejemplos de a uno, ingresamos N a la red y buscamos minimizar la función de costo del lote.
- La función de costo será el ECM

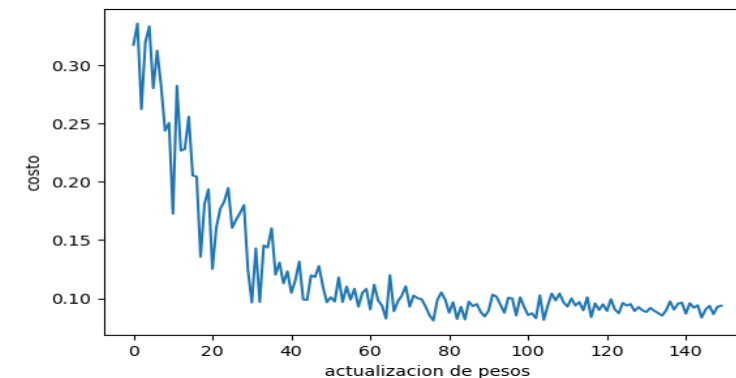
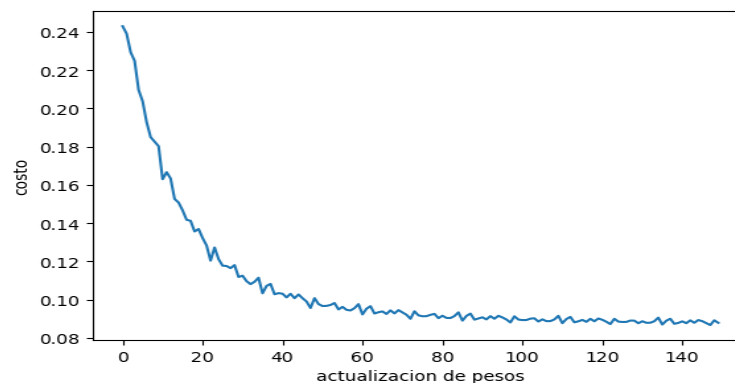
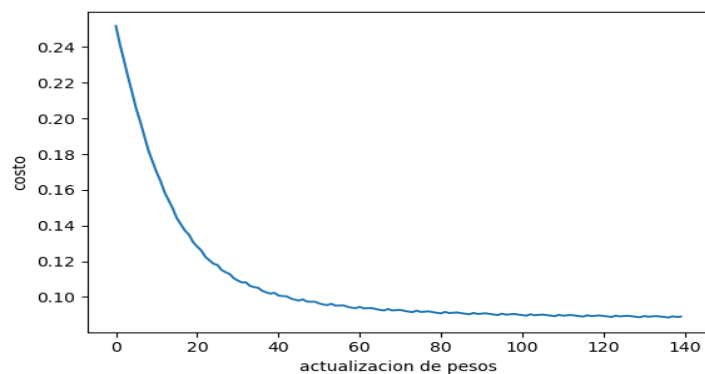
$$C = \frac{1}{N} \sum_{i=1}^N (d_i - f(neta_i))^2$$

N es la cantidad de ejemplos que conforman el lote.



DESCENSO DE GRADIENTE

Batch	Mini-batch	Stochastic
Ingresa TODOS los ejemplos y luego actualiza los pesos.	Ingresa un LOTE de N ejemplos y luego actualiza los pesos	Ingresa UN ejemplo y luego actualiza los pesos
$C = \frac{1}{M} \sum_{i=1}^M (d_i - f(neta_i))^2$	$C = \frac{1}{N} \sum_{i=1}^N (d_i - f(neta_i))^2 \quad N \ll M$	$C = (d - f(neta))^2$



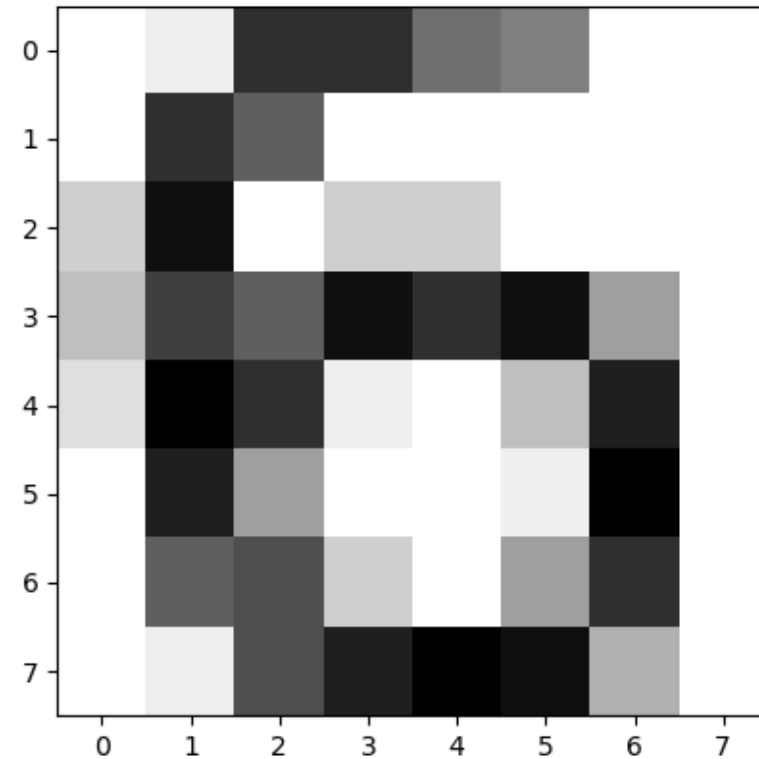
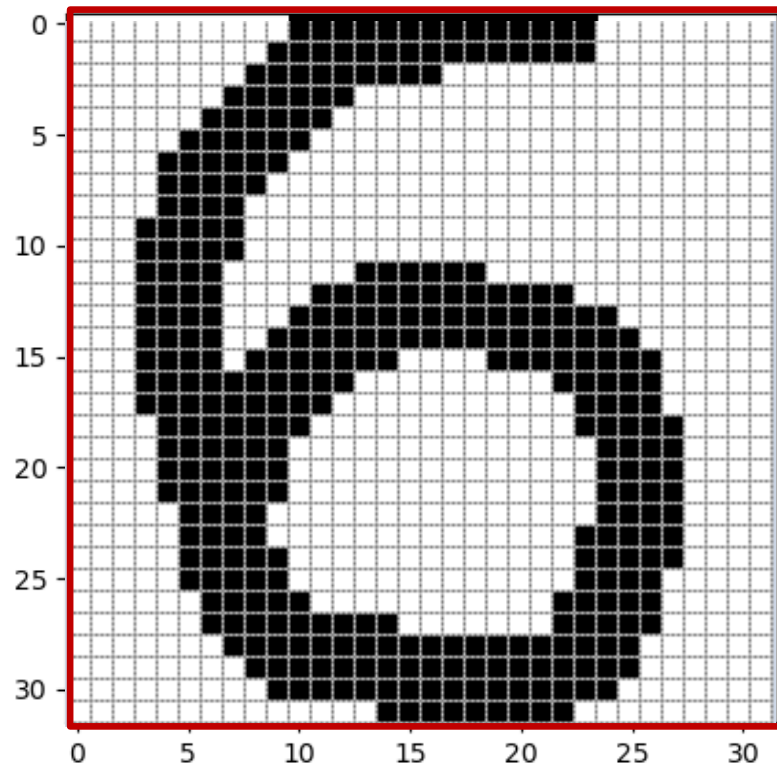
RECONOCEDOR DE DÍGITOS ESCRITOS A MANO

- Se desea entrenar un multiperceptrón para reconocer dígitos escritos a mano. Para ello se dispone de los mapas de bits correspondientes a 3823 dígitos escritos a mano por 30 personas diferentes en el archivo “optdigits_train.csv”.
- El desempeño de la red será probado con los dígitos del archivo “optdigits_test.csv” escritos por otras 13 personas.



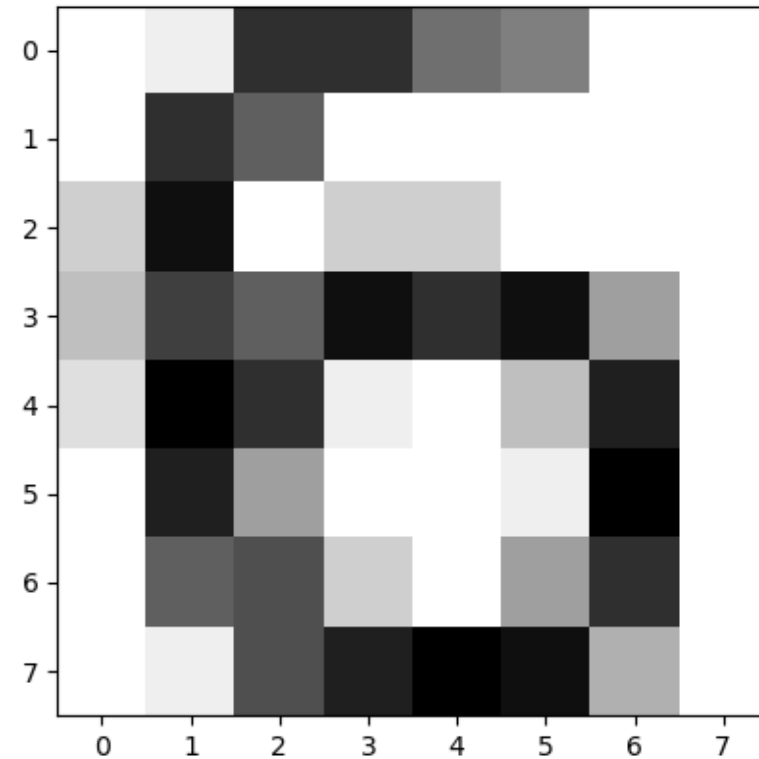
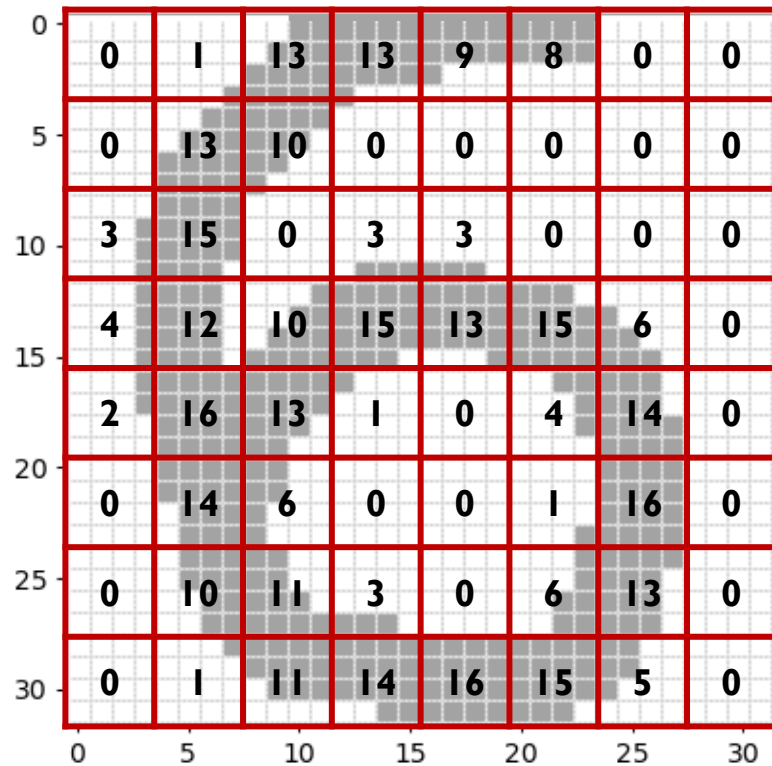
“OPTDIGITS_TRAIN.CSV” Y “OPTDIGITS_TEST.CSV”

- Cada dígito está representado por una matriz numérica de 8x8



“OPTDIGITS_TRAIN.CSV” Y “OPTDIGITS_TEST.CSV”

- Cada dígito está representado por una matriz numérica de 8x8



RN PARA RECONOCER DÍGITOS MANUSCRITOS

```
In [90]: print("ite = %d %% aciertos X_train : %.3f" % (ite,  
metrics.accuracy_score(Y_train,Y_pred)))  
ite = 200 %% aciertos X_train : 0.982
```

```
In [91]: MM = metrics.confusion_matrix(Y_train,Y_pred)  
....: print("Matriz de confusión TRAIN:\n%s" % MM)
```

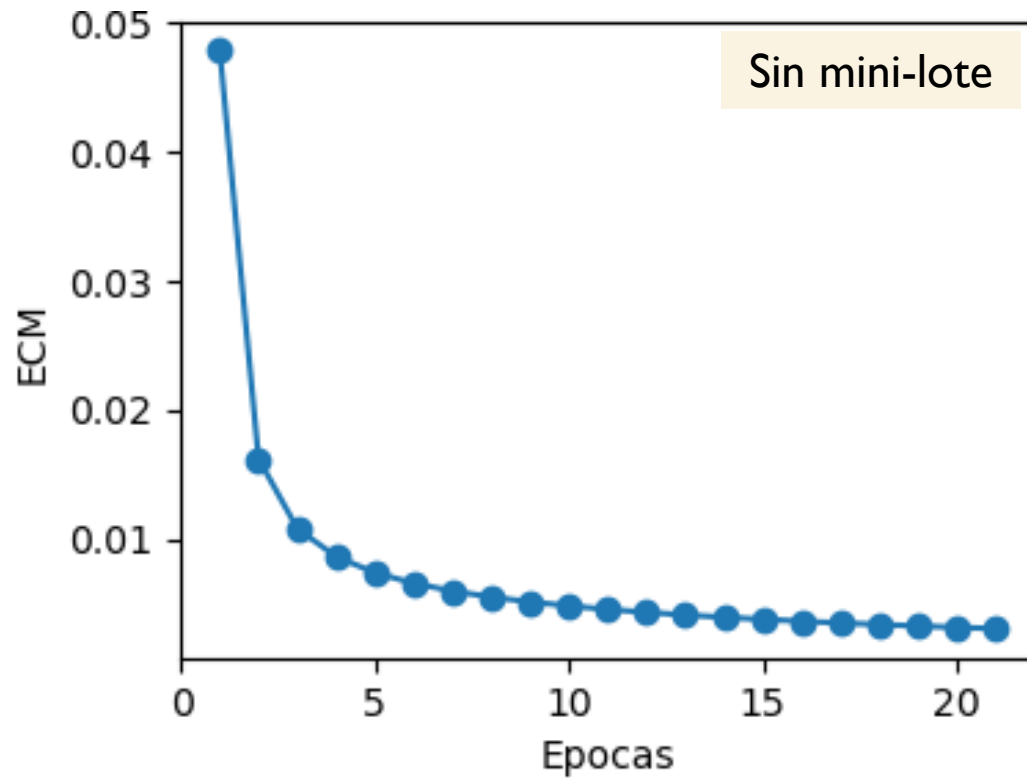
Matriz de confusión TRAIN:

```
[[375  0  0  0  0  0  0  0  1  0]  
 [ 7 382  0  0  0  0  0  0  0  0]  
 [ 2  0 378  0  0  0  0  0  0  0]  
 [ 7  0  0 380  0  2  0  0  0  0]  
 [ 3  0  0  0 383  0  1  0  0  0]  
 [ 6  0  0  1  0 369  0  0  0  0]  
 [ 2  2  0  0  0  0 373  0  0  0]  
 [ 0  0  0  1  0  0  0 386  0  0]  
 [17  2  0  0  0  0  0  0 361  0]  
 [13  1  0  1  0  1  0  0  0 366]]
```

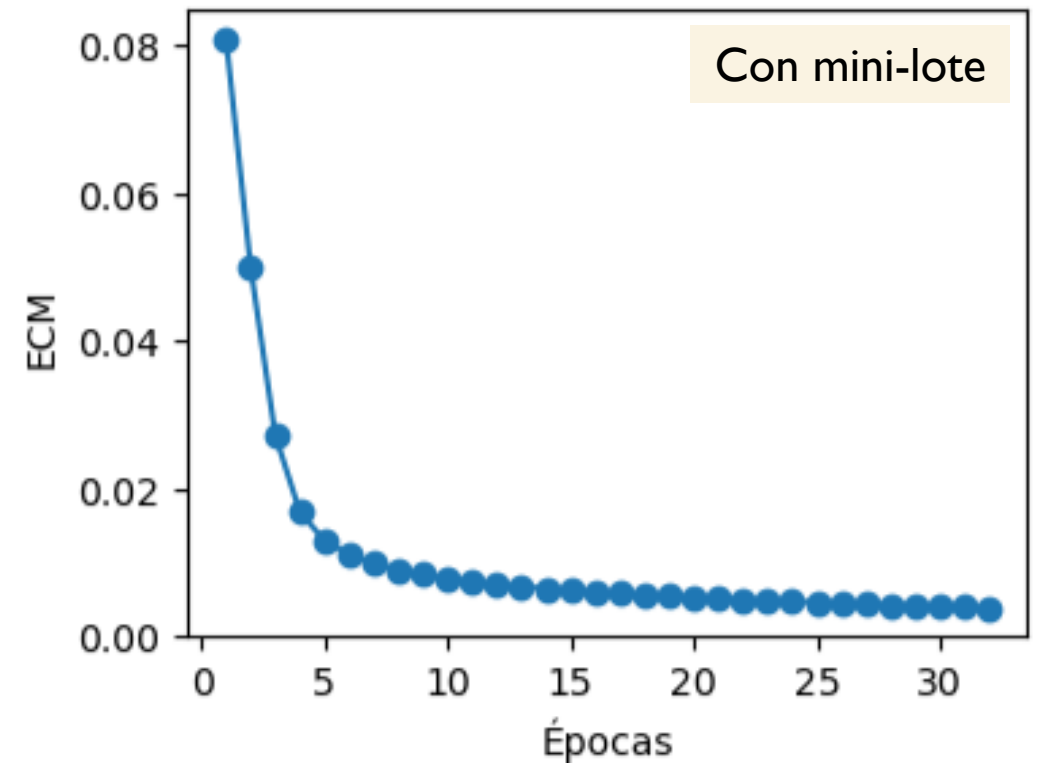
El tiempo de
entrenamiento es menor si
se usa mini-lote

MLP_MNIST_8x8.ipynb
MLP_MNIST_8x8_miniLote.ipynb

RECONOCIMIENTO DE DÍGITOS CON Y SIN MINILOTE




nEj=3823, Épocas=21, iteraciones=80283
duración 3.2125061 seg

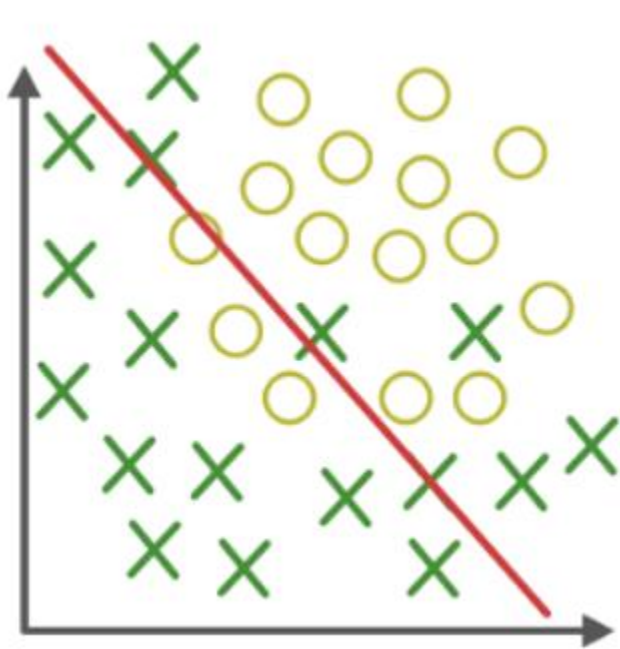


Lote=150, nLotes=25, épocas=32, iteraciones=800
duración 0.1465712 seg

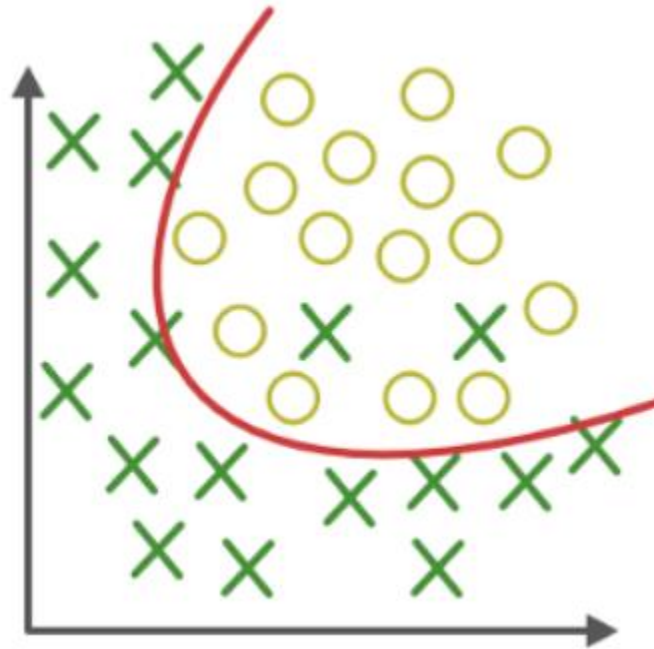
TÉCNICAS DE OPTIMIZACIÓN

- Descenso de gradiente estocástico (SGD) y el uso de mini-lotes
- Capacidad de generalización de la red - Sobreajuste 
- Mejoras introducidas
 - Momento: utiliza información de los gradientes anteriores
 - RMSProp: considera distintas magnitudes de cambio para reducir oscilaciones
 - Adam: combina los dos anteriores. Es el más usado.

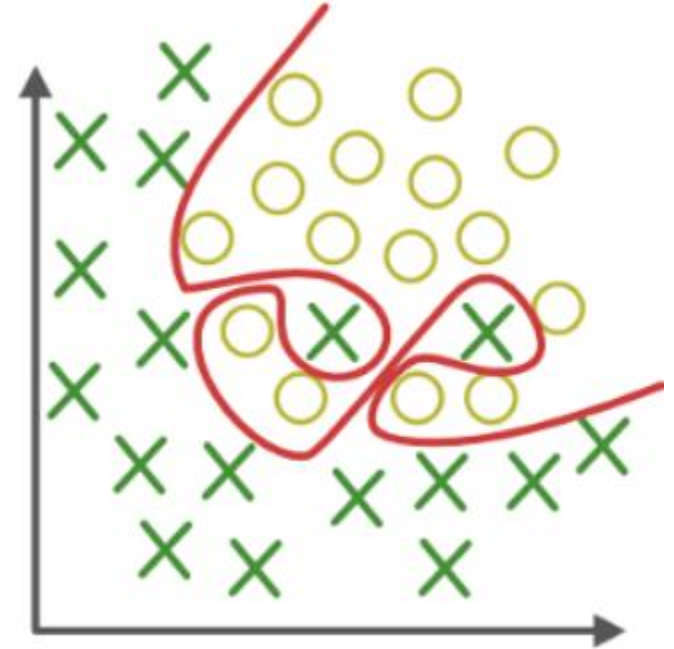
CAPACIDAD DE GENERALIZACIÓN DE LA RED



Underfitting
(demasiado simple)

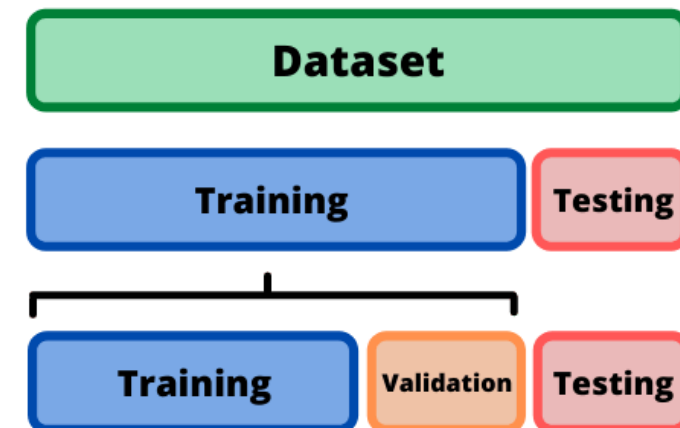
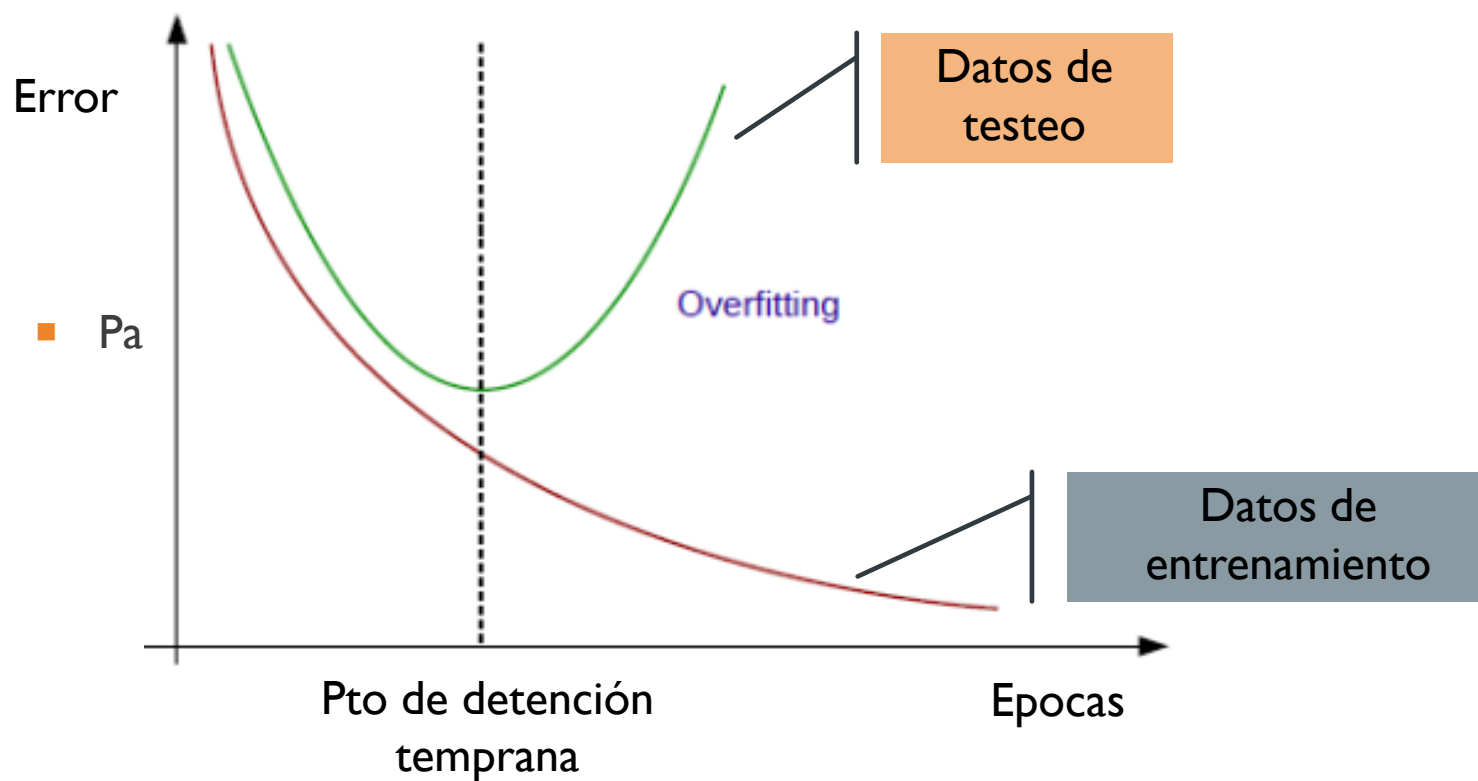


Generalización correcta



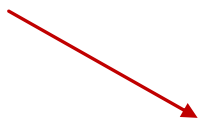
Overfitting
(demasiados parámetros)

SOBREAJUSTE



PARADA TEMPRANA

```
from keras.callbacks import EarlyStopping  
model = ...  
model.compile( ... )  
  
es = EarlyStopping(monitor='val_accuracy', patience=30, min_delta=0.0001)  
H = model.fit(x = X_train, y = Y_train, epochs=4000, batch_size = 20,  
              validation_data = (X_test, Y_test), callbacks=[es])  
  
print("Epocas = %d" % es.stopped_epoch)
```



validation_split=0.2

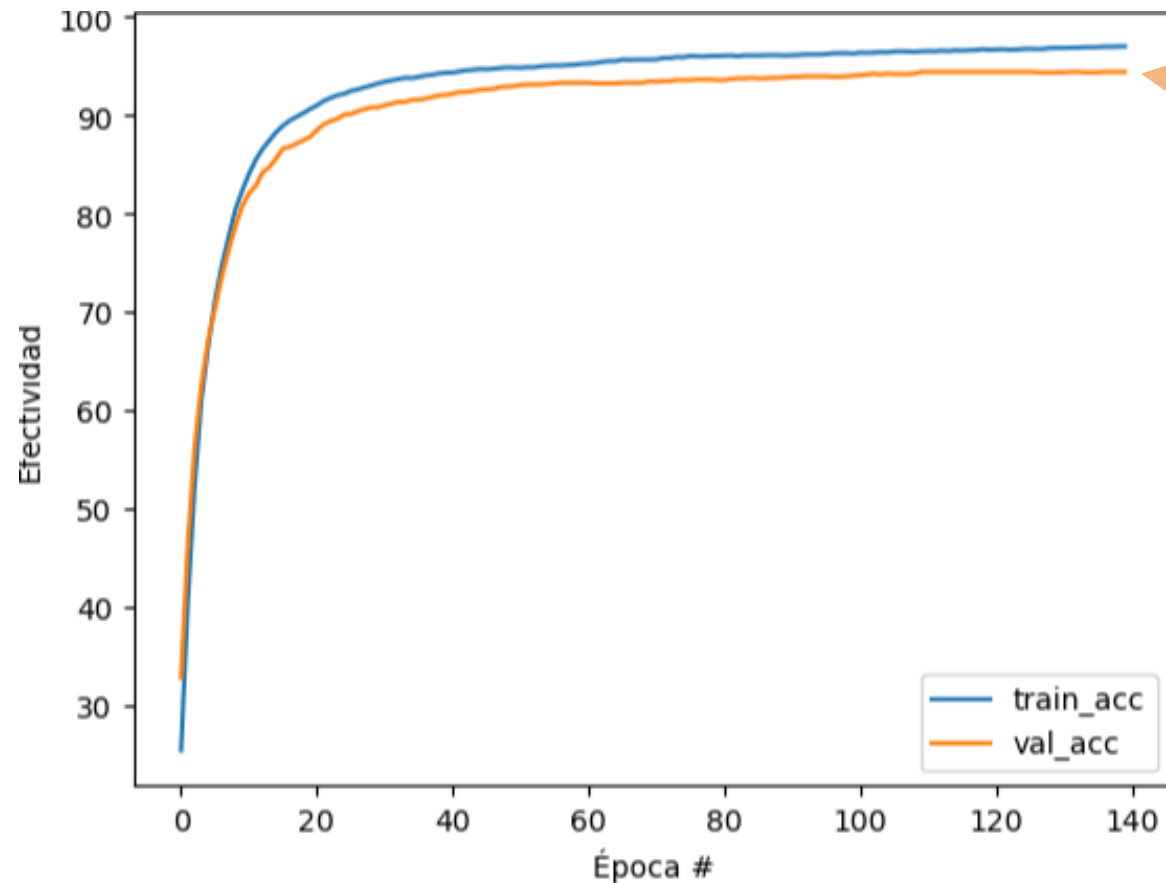
Keras_MNIST.ipynb

EARLYSTOPPING

- Detiene el entrenamiento cuando una métrica ha dejado de mejorar.
- Parámetros principales
 - **monitor**: valor a monitorear
 - **min_delta**: un cambio absoluto en el valor monitoreado inferior a min_delta, se considerará como que no hubo mejora.
 - **patience**: Número de épocas sin mejora tras las cuales se detendrá el entrenamiento.
 - **modo**: Uno de {"auto", "min", "max"}. En el modo "min", el entrenamiento se detendrá cuando el valor monitoreado haya dejado de disminuir; en el modo "max" se detendrá cuando el valor monitoreado haya dejado de aumentar; en el modo "auto", la dirección se infiere automáticamente del nombre del valor monitoreado.
 - **restore_best_weights**: Si se restauran los pesos del modelo de la época con el mejor resultado del valor monitoreado.

https://keras.io/api/callbacks/early_stopping/

EVOLUCIÓN DEL ENTRENAMIENTO



monitor='val_accuracy'
patience=30
min_delta=0.0001

Keras_MNIST.ipynb

REDUCCIÓN DEL SOBREAJUSTE

- Si lo que se busca es reducir el sobreajuste puede probar
 - Incrementar la cantidad de ejemplos de entrenamiento.
 - Reducir la complejidad del modelo, es decir usar menos pesos (menos capas o menos neuronas por capa).
 - Aplicar una técnica de regularización
 - Regularización L2
 - Regularización L1
 - Dropout

Tienen por objetivo que los pesos de la red se mantengan pequeños

SOBREAJUSTE - REGULARIZACIÓN L2

- También conocida como técnica de decaimiento de pesos

$$C = C_o + \frac{\lambda}{2} \sum_k w_k^2$$

donde C_o es la función de costo original sin regularizar

- La derivada de la función de costo regularizada será

$$\frac{\partial C}{\partial w_k} = \frac{\partial C_o}{\partial w_k} + \lambda w_k$$

SOBREAJUSTE - REGULARIZACIÓN L2

Función de costo regularizada

$$C = C_o + \frac{\lambda}{2} \sum_k w_k^2$$

Derivada

$$\frac{\partial C}{\partial w_k} = \frac{\partial C_o}{\partial w_k} + \lambda w_k$$

- Actualización de los pesos

$$w_k = w_k - \alpha \frac{\partial C_o}{\partial w_k} - \lambda w_k$$

$$w_k = (1 - \lambda) w_k - \alpha \frac{\partial C_o}{\partial w_k}$$

SOBREAJUSTE - REGULARIZACIÓN L1

Función de costo regularizada

$$C = C_o + \lambda \sum_k |w_k|$$

SOBREAJUSTE - REGULARIZACIÓN L1

Función de costo regularizada

$$C = C_o + \lambda \sum_k |w_k|$$

Derivada

$$\frac{\partial C}{\partial w_k} = \frac{\partial C_o}{\partial w_k} + \lambda \operatorname{sign}(w_k)$$

SOBREAJUSTE - REGULARIZACIÓN L1

Función de costo regularizada

$$C = C_o + \lambda \sum_k |w_k|$$

Derivada

$$\frac{\partial C}{\partial w_k} = \frac{\partial C_o}{\partial w_k} + \lambda \operatorname{sign}(w_k)$$

- Actualización de los pesos

$$w_k = w_k - \alpha \frac{\partial C_o}{\partial w_k} - \lambda \operatorname{sign}(w_k)$$

LI VS L2

Regularización L1

- **Lleva los pesos a 0:** útil para que el modelo ignore características irrelevantes.
- **Selección automática de características:** ideal para datos con muchas variables donde solo unas pocas son relevantes.
- **Aplicaciones:** Modelos de alta dimensionalidad (por ejemplo, compresión de modelos, selección de características).

Regularización L2

- **Mantiene todos los pesos pequeños:** pero no los hace exactamente 0.
- **Mejor generalización:** útil cuando se espera que todas las características sean relevantes.
- **Aplicaciones:** Modelos profundos, evitar sobreajuste en redes neuronales complejas.

KERAS.REGULARIZERS

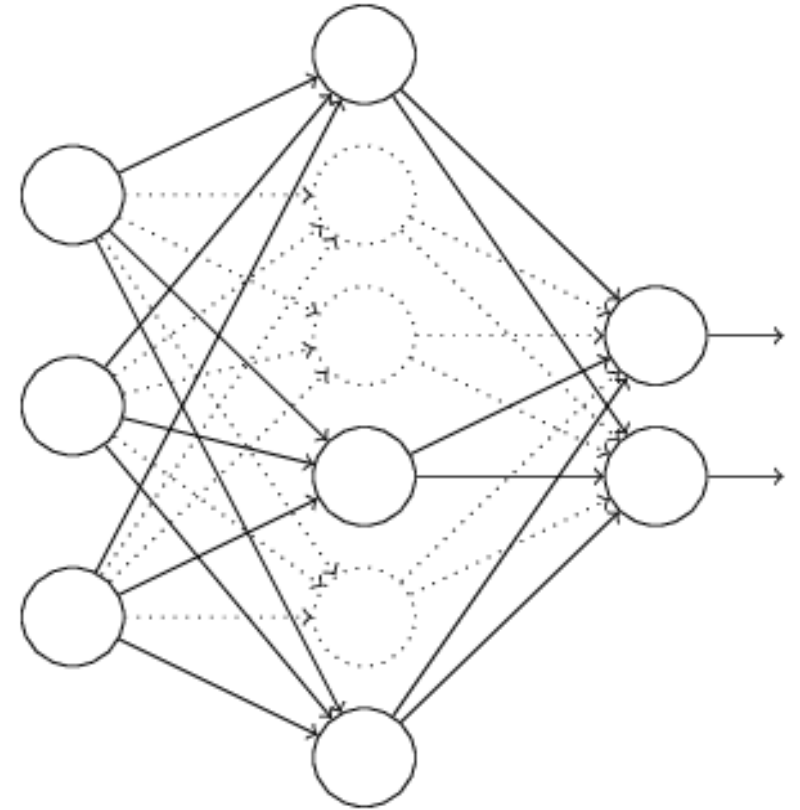
```
from keras.layers import Dense
from keras.regularizers import l2, l1, l1_l2
...
model.add(Dense(32, kernel_regularizer=l2(0.01),
                bias_regularizer=l2(0.01)))
```

- Se pueden aplicar ambos

```
model.add(Dense(32, kernel_regularizer=l1_l2(l1=0.01, l2=0.01),
                bias_regularizer=l1_l2(l1=0.01, l2=0.01)))
```

SOBREAJUSTE - DROPOUT

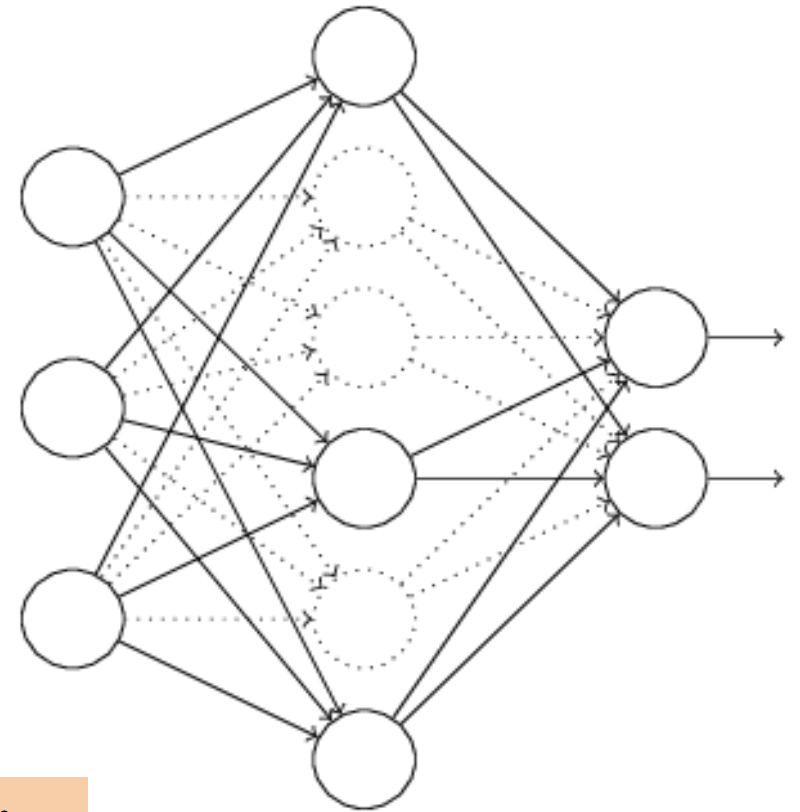
- No modifica la función de costo sino la arquitectura de la de la red.
- Proceso
 - Selecciona aleatoriamente las neuronas que no participarán en la próxima iteración y las “borra” temporalmente.
 - Actualiza los pesos (del mini lote si corresponde).
 - Restaura las neuronas “borradas”.
 - Repite hasta que se estabilice.



KERAS DROPOUT

```
from keras.layers import Dense
from keras.layers import Dropout
...
model.add(Dense(6, input_shape=[3]))
model.add(Dropout(0.5))
model.add(Dense(2))
```

**Probabilidad de anular cada entrada de la capa anterior
En este caso el 50% de las entradas serán anuladas**

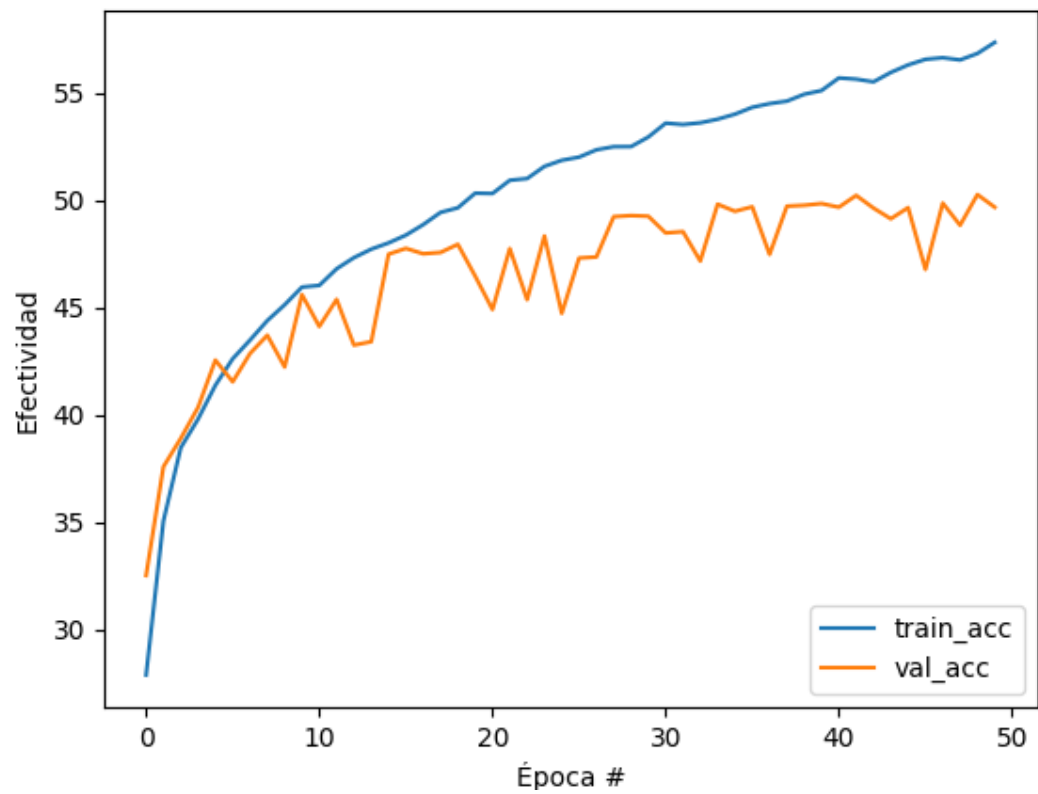


CIFAR-10

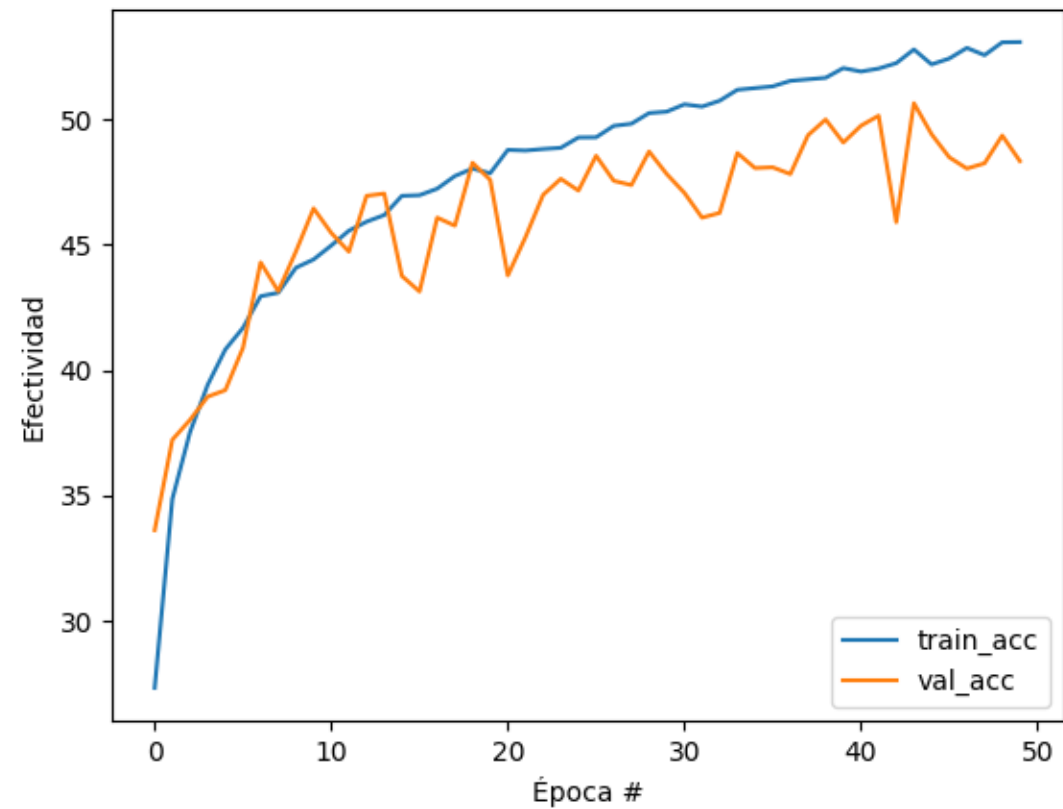


- Se compone de 60.000 imágenes de 32x32x3, en espacio RGB.
- Hay 50.000 imágenes de entrenamiento y 10.000 imágenes de prueba.
- Hay 10 clases, donde cada una está representada por 6.000 imágenes.
- Las clases son mutuamente excluyentes

CIFAR-10



Keras_CIFAR10_softmax.ipynb



Keras_CIFAR10_softmax_L2.ipynb

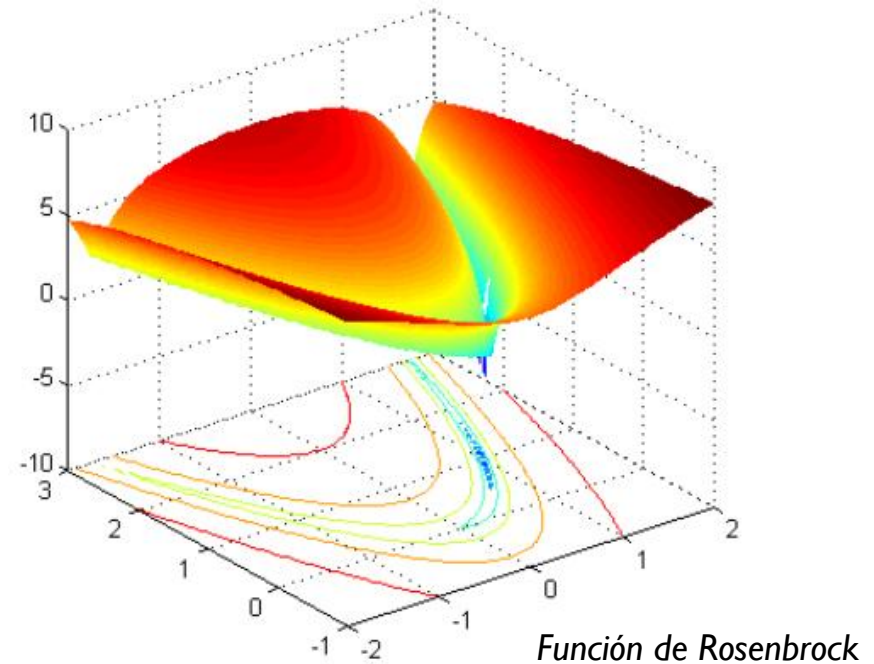
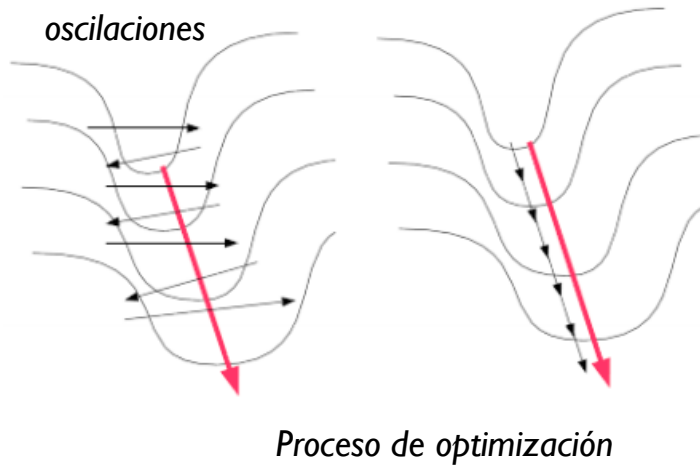
TÉCNICAS DE OPTIMIZACIÓN

- Descenso de gradiente estocástico (SGD) y el uso de mini-lotes
- Capacidad de generalización de la red - Sobreajuste
- Mejoras introducidas
 - Momento: utiliza información de los gradientes anteriores
 - RMSProp: considera distintas magnitudes de cambio para reducir oscilaciones
 - Adam: combina los dos anteriores. Es el más usado.



DESCENSO DE GRADIENTE

- El descenso de gradiente es un proceso lento porque presenta oscilaciones

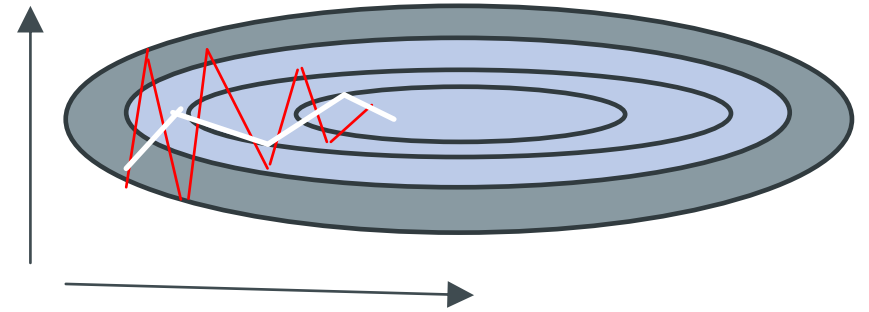


- Para resolverlo, en lugar de utilizar directamente el valor del gradiente se trabaja con la **media de gradientes anteriores ponderada de manera exponencial**

SGD CON MOMENTO

$$v_t = \beta v_{t-1} + (1 - \beta)(\nabla C)_t$$

$$w_t = w_t - \alpha v_t$$



- Las modificaciones sobre W tienen en cuenta el promedio de los gradientes anteriores.
- La cantidad de gradientes anteriores a considerar son aprox. $\frac{1}{1-\beta}$
- Esto reduce las oscilaciones.

SGD CON MOMENTO

$$v_t = \beta v_{t-1} + (1 - \beta)(\nabla C)_t$$

- Usemos $\beta = 0.9$ en la iteración $t = 10$

$$v_{10} = 0.9 * v_9 + (1 - 0.9)(\nabla C)_{10}$$

SGD CON MOMENTO

$$v_t = \beta v_{t-1} + (1 - \beta)(\nabla C)_t$$

- Usemos $\beta = 0.9$ en la iteración $t = 10$

$$v_{10} = 0.9 * v_9 + (1 - 0.9)(\nabla C)_{10} = 0.1 \nabla C_{10} + 0.9 v_9$$

SGD CON MOMENTO

$$v_t = \beta v_{t-1} + (1 - \beta)(\nabla C)_t$$

- Usemos $\beta = 0.9$ en la iteración $t = 10$

$$v_{10} = 0.9 * v_9 + (1 - 0.9)(\nabla C)_{10} = 0.1 \nabla C_{10} + 0.9 v_9$$

$$v_{10} = 0.1 \nabla C_{10} + 0.9 (0.1 \nabla C_9 + 0.9 v_8)$$

$$v_{10} = 0.1 \nabla C_{10} + 0.1 * 0.9 \nabla C_9 + 0.9^2 v_8$$

$$v_{10} = 0.1 \nabla C_{10} + 0.1 * 0.9 \nabla C_9 + 0.9^2 (0.9 v_7 + 0.1 \nabla C_8)$$

SGD CON MOMENTO

$$v_t = \beta v_{t-1} + (1 - \beta)(\nabla C)_t$$

- Usemos $\beta = 0.9$ en la iteración $t = 10$

$$v_{10} = 0.9 * v_9 + (1 - 0.9)(\nabla C)_{10} = 0.1 \nabla C_{10} + 0.9 v_9$$

$$v_{10} = 0.1 \nabla C_{10} + 0.9 (0.1 \nabla C_9 + 0.9 v_8)$$

$$v_{10} = 0.1 \nabla C_{10} + 0.1 * 0.9 \nabla C_9 + 0.9^2 v_8$$

$$v_{10} = 0.1 \nabla C_{10} + 0.1 * 0.9 \nabla C_9 + 0.1 * 0.9^2 \nabla C_8 + 0.9^3 v_7 + \dots$$

La cantidad de gradientes anteriores a considerar son aprox. $\frac{1}{1-\beta}$ \therefore si $\beta=0.9$ serán aprox. 10

SGD CON MOMENTO

$V_w = 0$

$V_b = 0$

for t in range(iteraciones):

 Calcular gradientes ∇_w y ∇_b

$V_w = \text{beta} * V_w + (1-\text{beta}) * \nabla_w$

$V_b = \text{beta} * V_b + (1-\text{beta}) * \nabla_b$

$W = W - \text{alfa} * V_w$

$b = b - \text{alfa} * V_b$

`keras.optimizers.SGD(learning_rate=0.01, momentum=0.9)`

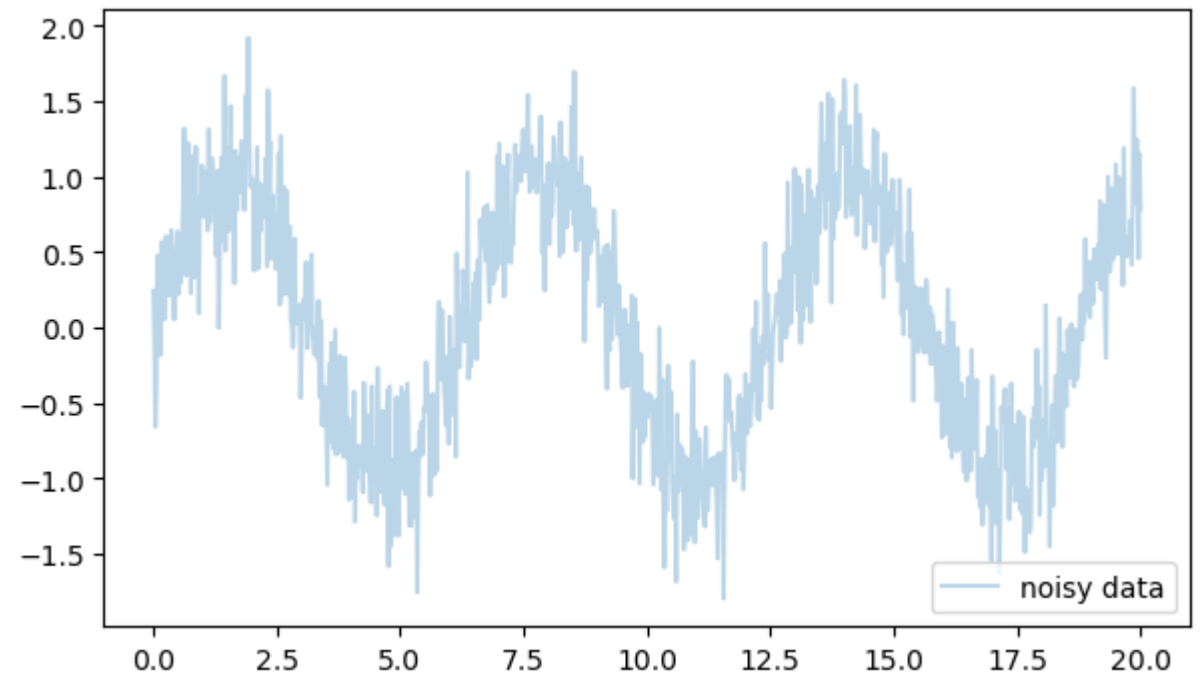
MEDIAS PONDERADAS EXPONENCIALMENTE

Exponentially weighted averages (EWA)

```
x = np.linspace(0.0, 1.0, num=1000) * 10  
noise = np.random.normal(scale=0.3, size=len(x))  
y = np.sin(x) + noise
```

```
beta = 0.9  
v = 0.0  
ewa90 = []  
for t in y:  
    v = beta * v + (1-beta) * t  
    ewa90.append(v)
```

Medias_EWA.ipynb



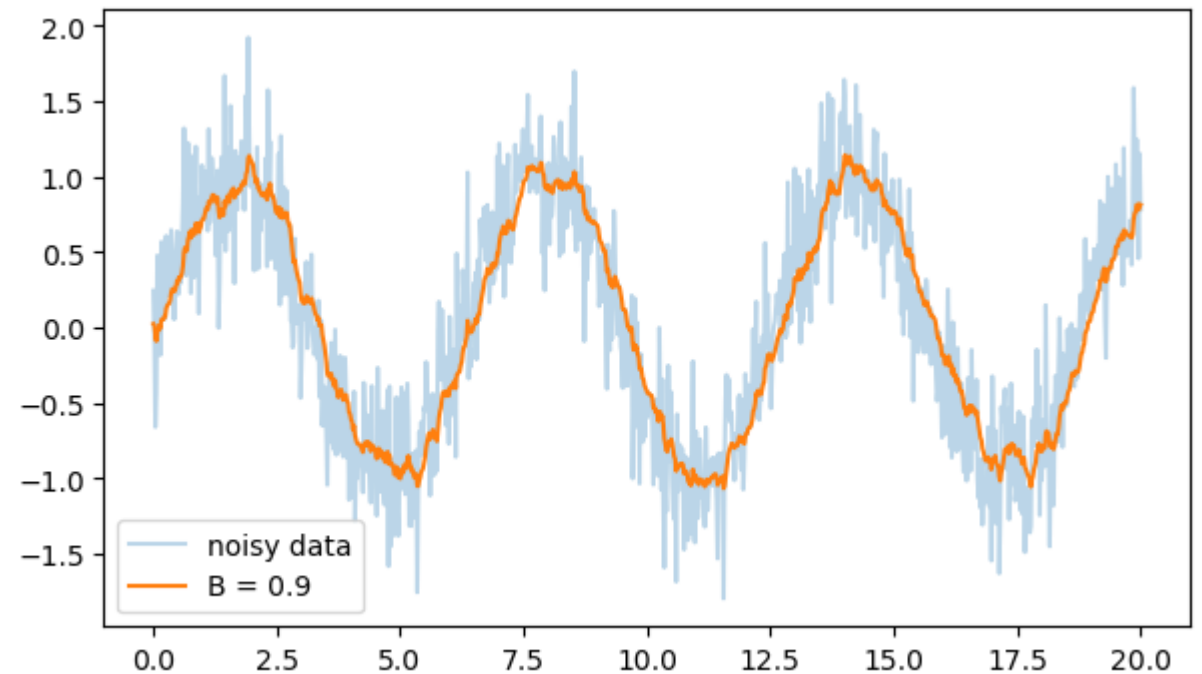
PROMEDIO PONDERADO EXPONENCIALMENTE

Exponentially weighted average (EWA)

```
x = np.linspace(0.0, 1.0, num=1000) * 10  
noise = np.random.normal(scale=0.3, size=len(x))  
y = np.sin(x) + noise
```

```
beta = 0.9  
v = 0.0  
ewa90 = []  
for t in y:  
    v = beta * v + (1-beta) * t  
    ewa90.append(v)
```

Medias_EWA.ipynb



PROMEDIO PONDERADO EXPONENCIALMENTE

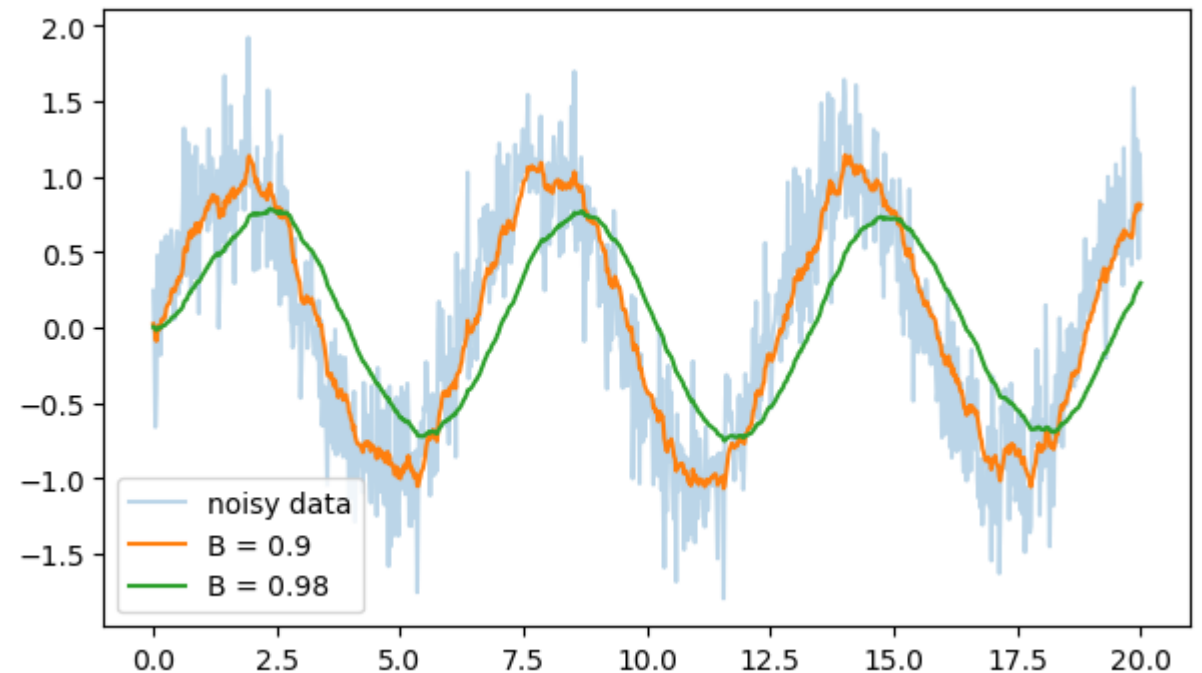
Exponentially weighted average (EWA)

```
x = np.linspace(0.0, 1.0, num=1000) * 10
noise = np.random.normal(scale=0.3, size=len(x))
y = np.sin(x) + noise
```

```
beta = 0.9
v = 0.0
ewa90 = []
for t in y:
    v = beta * v + (1-beta) * t
    ewa90.append(v)
```

```
beta = 0.98
v = 0.0
ewa98 = []
for t in y:
    v = beta * v + (1-beta) * t
    ewa98.append(v)
```

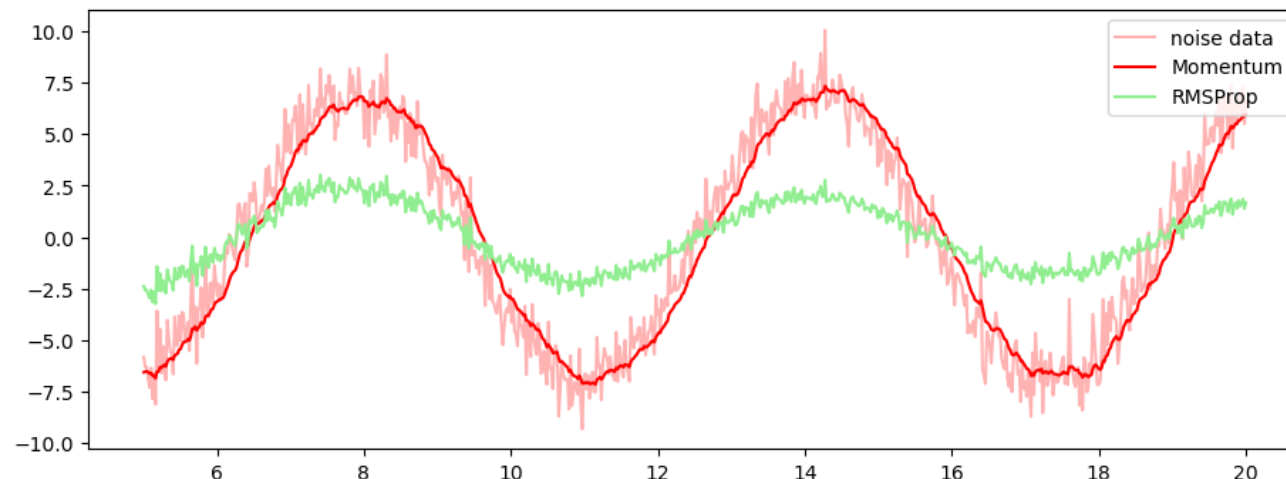
Medias_EWA.ipynb



RMSPROP

$$s = \beta s + (1 - \beta) (\nabla C)^2$$

$$w = w - \alpha \frac{\nabla C}{\sqrt{s + \varepsilon}}$$



- Las modificaciones sobre w tienen en cuenta el promedio de los gradientes anteriores.
- Las modificaciones más grandes serán divididas por coeficientes más grandes; por lo tanto se reducen.
- Las modificaciones más chicas se incrementan.

Dependiendo del problema puede ser más eficiente que SGD+Momento

RMSPROP

```
from keras.optimizers import RMSprop

X,Y = cargar_datos()

model = Sequential()
model.add(...)

model.compile(
    loss='categorical_crossentropy',
    optimizer = RMSprop(learning_rate=0.001),
    metrics=['accuracy'])

model.fit(X,Y, epochs=10, batch_size=32)
```


ADAM

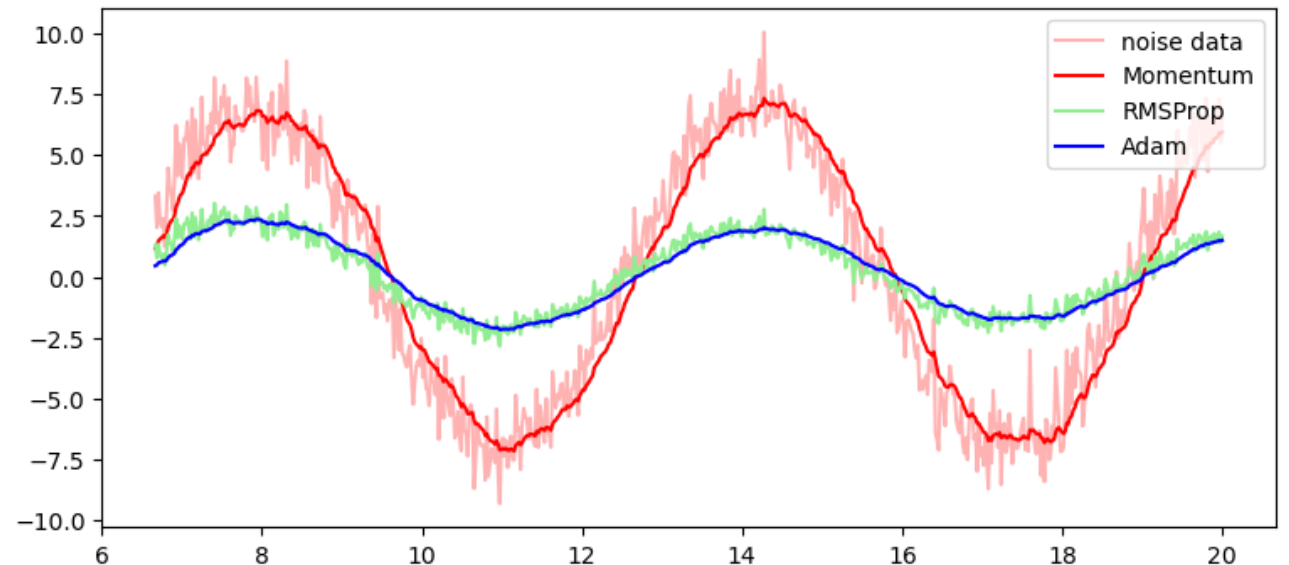
<https://keras.io/api/optimizers/adam/>

- Combina momento y RMSprop

$$v = \beta_1 v + (1 - \beta_1) \nabla C$$

$$s = \beta_2 s + (1 - \beta_2) (\nabla C)^2$$

$$w = w - \alpha \frac{v}{\sqrt{s + \epsilon}}$$




- Los valores recomendados son $\beta_1 = 0.9$ y $\beta_2 = 0.999$

```
model.compile(optimizer='adam', loss='mse')
```

RESUMEN

Resolución de una tarea de clasificación

- Conjunto de datos etiquetados (aprendizaje supervisado)
- Definición de la arquitectura de la red
 - Número de capas y tamaño de cada una
 - Función de activación a usar en cada capa
- Entrenamiento
 - Función de error
 - Técnica de optimización para reducir el error
- Evaluar el modelo 

EVALUACIÓN DEL MODELO

- Matriz de confusión
- Métricas
 - Accuracy
 - Precisión
 - Recall
 - F1-score
 - AUC-ROC, AUC-PR

ROCA O MINA

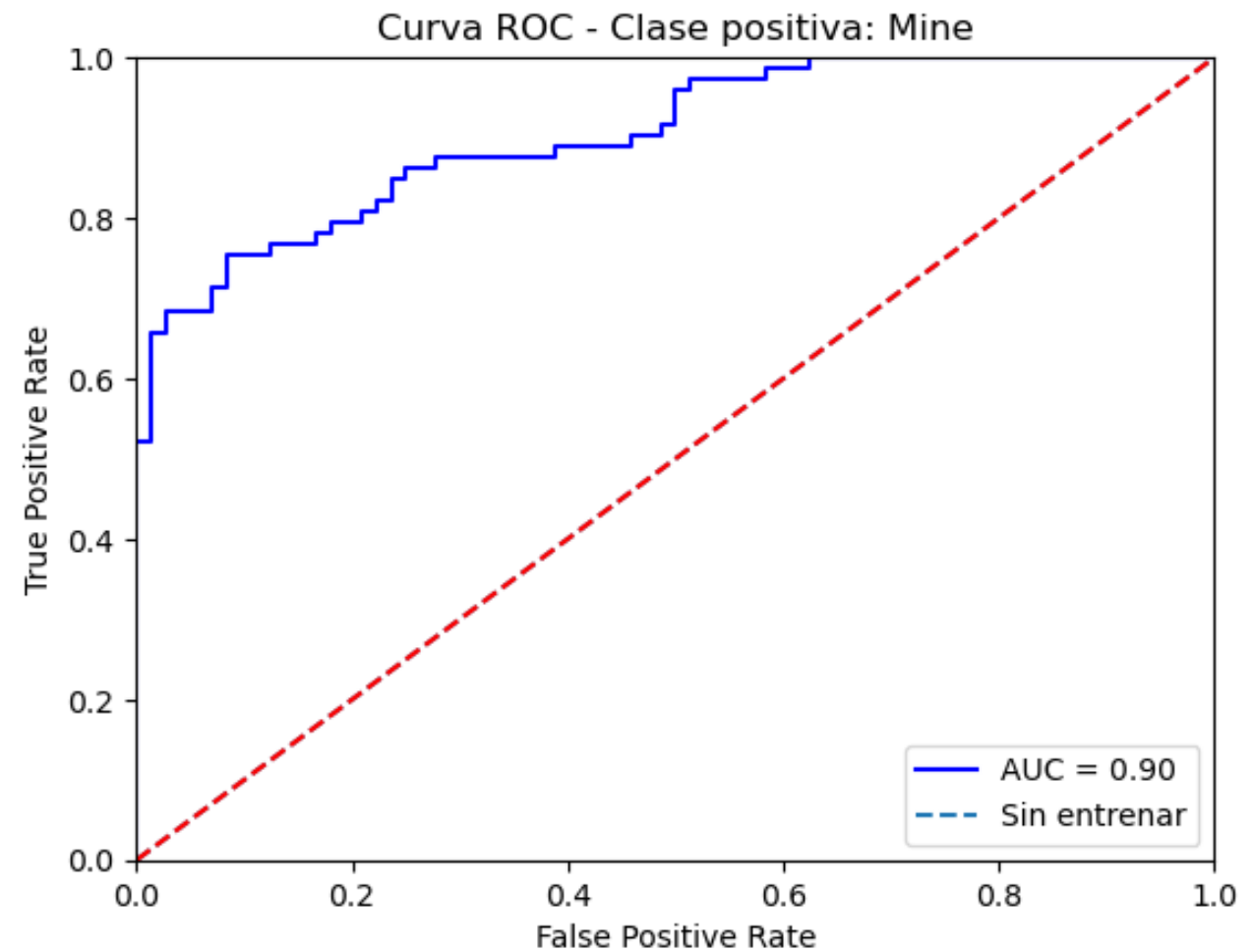
- A partir de los datos del archivo “Sonar.csv” se desea construir una red neuronal para discriminar entre señales de sonar rebotadas en un cilindro de metal (“Mine”) y aquellas rebotadas en una roca más o menos cilíndrica (“Rock”).
- Utilice el 70% para entrenar y el 30% para testear.
- Realice una evaluación del modelo obtenido

CURVA ROC

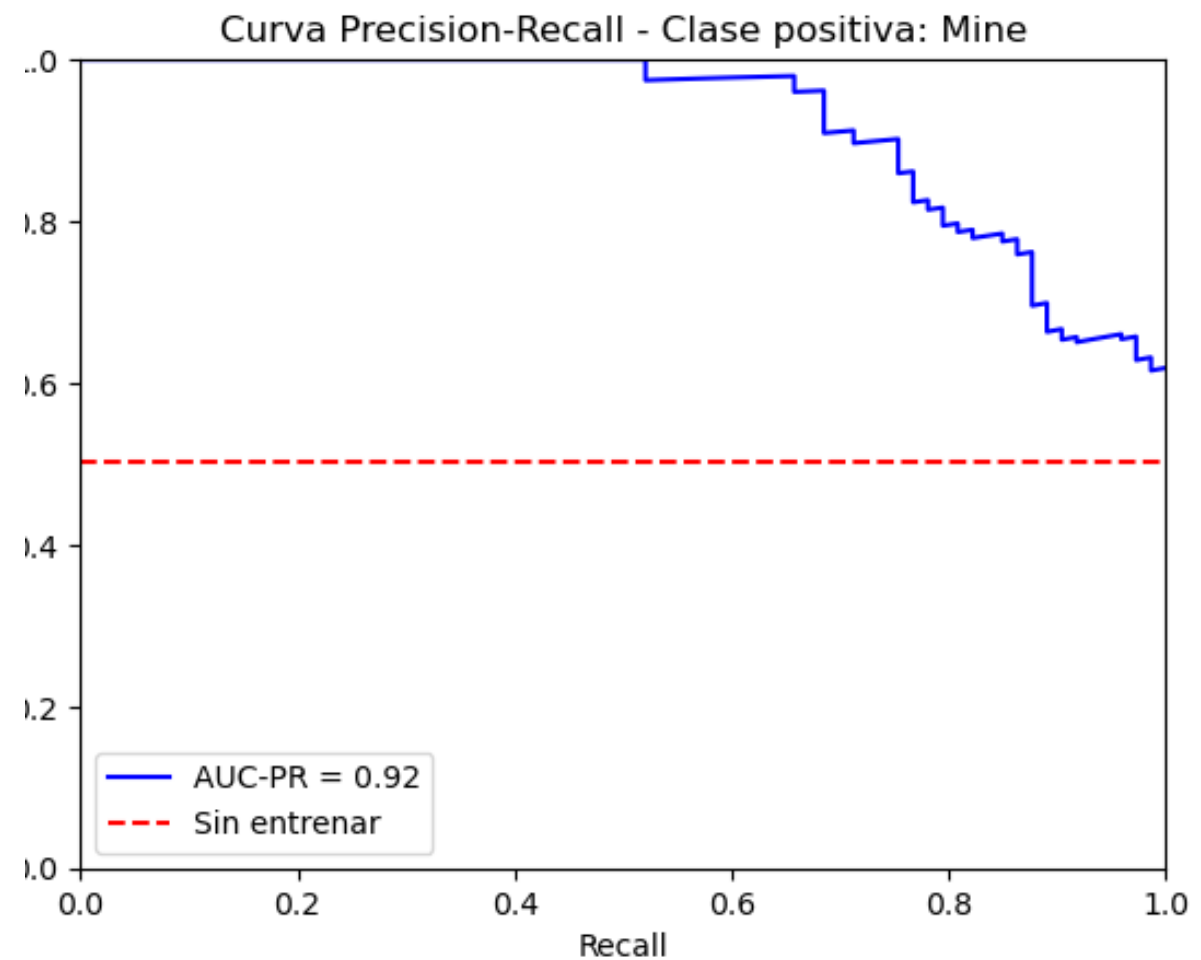
```
fpr, tpr, threshold = metrics.roc_curve(Y_true, Y_prob)
roc_auc = metrics.auc(fpr, tpr)

plt.figure()
plt.title('Receiver Operating Characteristic')
plt.plot(fpr, tpr, 'b', label = 'AUC = %0.2f' % roc_auc)
```

CURVA ROC



CURVA PR



FASHION MNIST

- El conjunto de datos Fashion MNIST (base de datos del Instituto Nacional de Normas y Tecnología de la Moda Modificada) está compuesto por 60 000 muestras del conjunto de entrenamiento y 10 000 muestras del conjunto de prueba. Cada muestra es una imagen en escala de grises de 28×28 con una etiqueta de una de las diez clases.
- A partir del notebook **Fashion_MNIST.ipynb** diseñe y entrene una red neuronal que sea capaz de clasificar los 10 tipos de prendas.
- Analice la performance del modelo

