Encuentro 23. Introducción a las redes neuronales

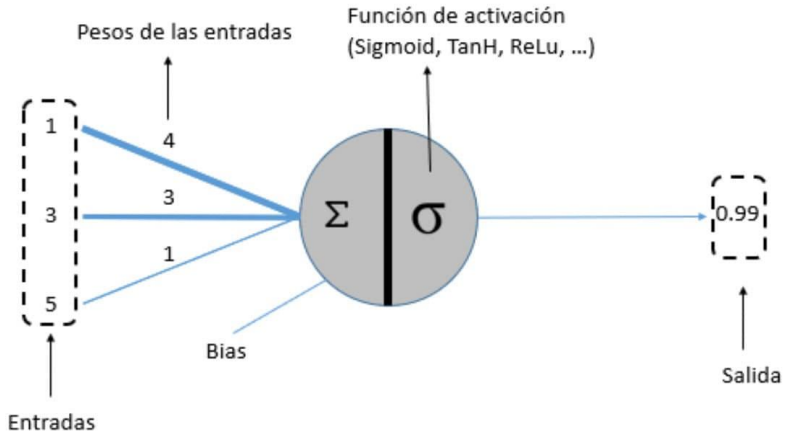
Las redes neuronales se utilizan para generar predicciones, análisis de texto, voz e imágenes y detección de objetos. En esta unidad veremos los conceptos básicos de las redes neuronales, cómo funcionan y cómo empezar a utilizarlas en un pequeño proyecto de clasificación.

Introducción

Las redes neuronales (NN) se enmarcan dentro del campo de la Inteligencia Artificial. Las NN nacieron en los años 60 pero no fueron utilizadas hasta décadas más adelante debido a la baja capacidad de cómputo y almacenamiento de la época. Sin embargo, con el crecimiento de las computadoras el interés por ellas aumentó exponencialmente.

Perceptrón simple

El Perceptrón simple se trata del modelo más sencillo de redes neuronales, ya que consta de una sola capa de neuronas con una única salida..



Este modelo recibe una o más entradas y realiza una suma ponderada para producir una salida. A dicha suma ponderada, se le añade un bias y se transmite a través de una función no lineal conocida como función de activación. Las funciones de activación suelen tener una forma sigmoide, pero también pueden adoptar la forma de otras funciones no lineales, funciones lineales a trozos o funciones de paso.

Perceptron multicapa (MLP)

El perceptrón multicapa es una red neuronal artificial formada por múltiples capas, de tal manera que tiene capacidad para resolver problemas no son linealmente separables, lo cual es la principal limitación del perceptrón simple. El perceptrón multicapa puede estar totalmente o localmente conectado. En el primer caso, cada salida de una neurona de la capa "i" es entrada de **todas** las neuronas de la capa "i+1", mientras que para el segundo

cada neurona de la capa "i" es entrada de un **conjunto** de neuronas (región) de la capa "i+1".

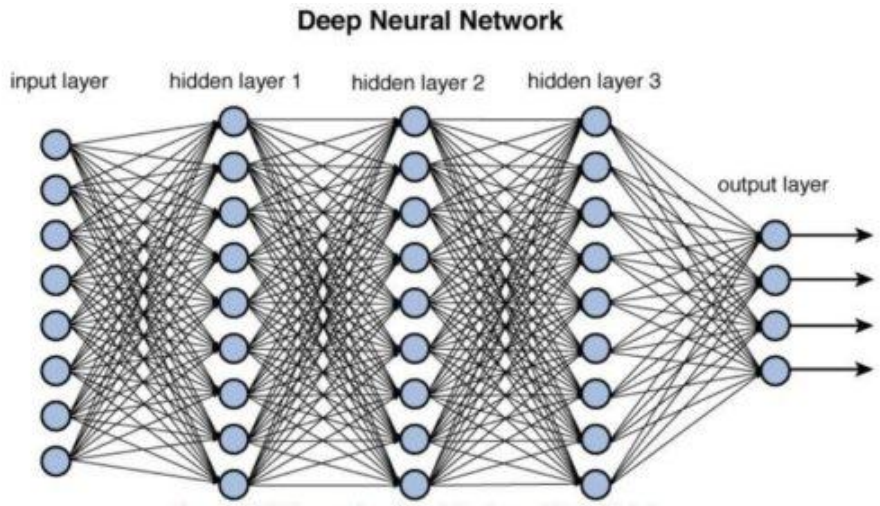
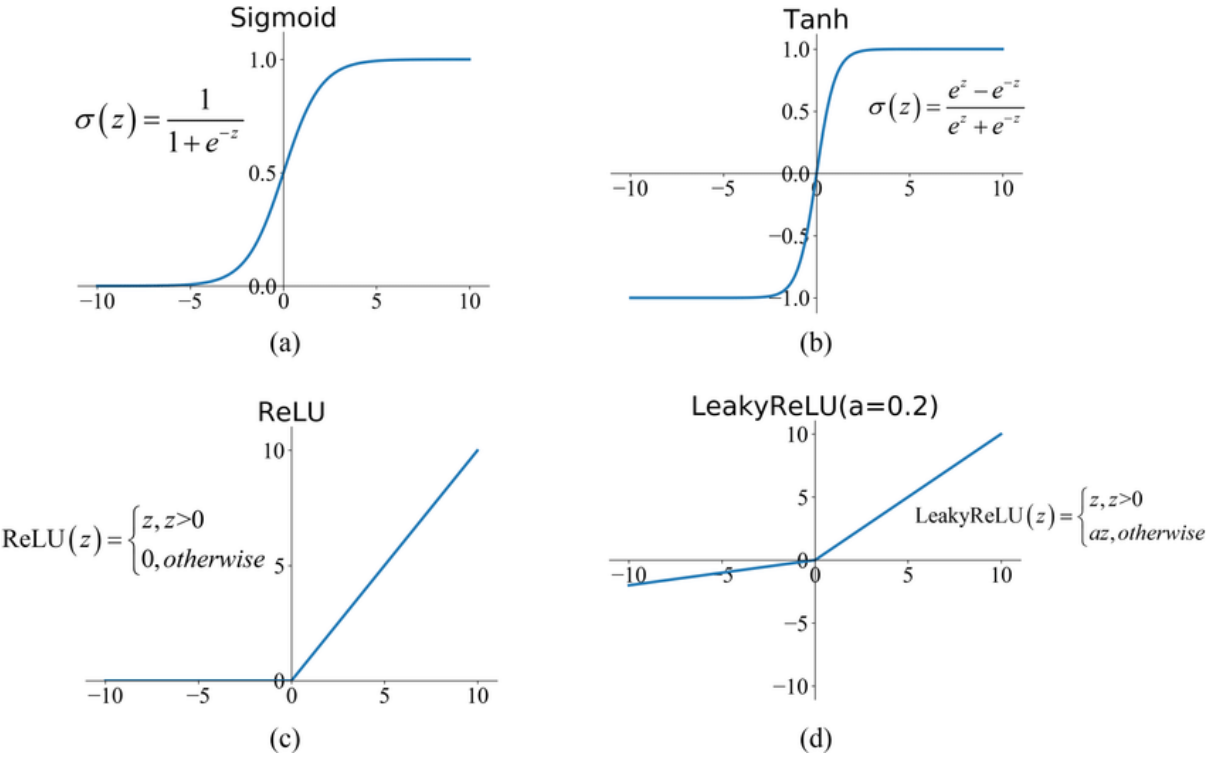


Figure 12.2 Deep network architecture with multiple layers.

Funciones de activación



Entrenamiento de una Red Neuronal.

Como dijimos anteriormente una red neuronal se compone de “neuronas” organizadas en capas. Cada neurona de la red (exceptuando la capa de entrada) es en realidad un sumatorio de todas sus entradas; que no son más que las salidas de las capas anteriores multiplicadas por unos pesos. A esta suma se le añade un término adicional llamado sesgo o bias. Y al resultado se le aplica una función no lineal conocida como función de activación.

Los parámetros de la red (pesos, bias) son precisamente los valores numéricos que trataremos de ajustar mediante entrenamiento usando un conjunto de muestras ya etiquetadas, como en cualquier otro problema de Machine Learning supervisado. El resultado final será un modelo que construido a partir de esos datos debería ser capaz de hacer predicciones con muestras futuras.

Inicialización de los parámetros

Para llevar a cabo el entrenamiento de nuestra red neuronal una vez elegida su arquitectura, lo primero que debemos hacer es inicializar sus parámetros. Si partimos de cero (sin un modelo pre-entrenado) es bastante común proceder de la siguiente forma:

- Inicializaremos los pesos aleatoriamente para ayudar a la red, rompiendo su simetría. El fin es evitar que todas las neuronas de una capa acaben aprendiendo lo mismo. Se suelen generar los pesos de cada capa usando una distribución normal de media cero y varianza 1/n ó 2/n (siendo n el número de entradas). Este valor de la varianza depende un poco de la función de activación que se coloque a la salida de la neurona (usaremos 2/n para ReLU).
- Inicializaremos los parámetros de bias a cero.

A partir de aquí procederemos iterativamente siguiendo un algoritmo de optimización, que tratará de minimizar la diferencia entre la salida real y la estimada por la red.

Descenso del gradiente

El algoritmo más utilizado para entrenar redes neuronales es el descenso del gradiente. ¿Y qué es eso del gradiente? Lo definiremos más adelante, pero de momento nos quedamos con la siguiente idea: el gradiente es un cálculo que nos permite saber cómo ajustar los parámetros de la red de tal forma que se minimice su desviación a la salida.

El algoritmo cuenta con varias versiones dependiendo del número de muestras que introduzcamos a la red en cada iteración:

- Descenso del gradiente en lotes (o batch): todos los datos disponibles se introducen de una vez. Esto supondrá problemas de estancamiento, ya que el gradiente se calculará usando siempre todas las muestras, y llegará un momento en que las variaciones serán mínimas. Como regla general: siempre nos conviene que la entrada a una red neuronal tenga algo de aleatoriedad.
- Descenso del gradiente estocástico: se introduce una única muestra aleatoria en cada iteración. El gradiente se calculará para esa muestra concreta, lo que supone la introducción de la deseada aleatoriedad, dificultando así el estancamiento. El problema de esta versión es su lentitud, ya que necesita de muchas más iteraciones, y además no aprovecha los recursos disponibles.
- Descenso del gradiente (estocástico) en mini-lotes (o mini-batch): en lugar de alimentar la red con una única muestra, se introducen N muestras en cada iteración; conservando las ventajas de la segunda versión y consiguiendo además que el entrenamiento sea más rápido debido a la paralelización de las operaciones. Nos

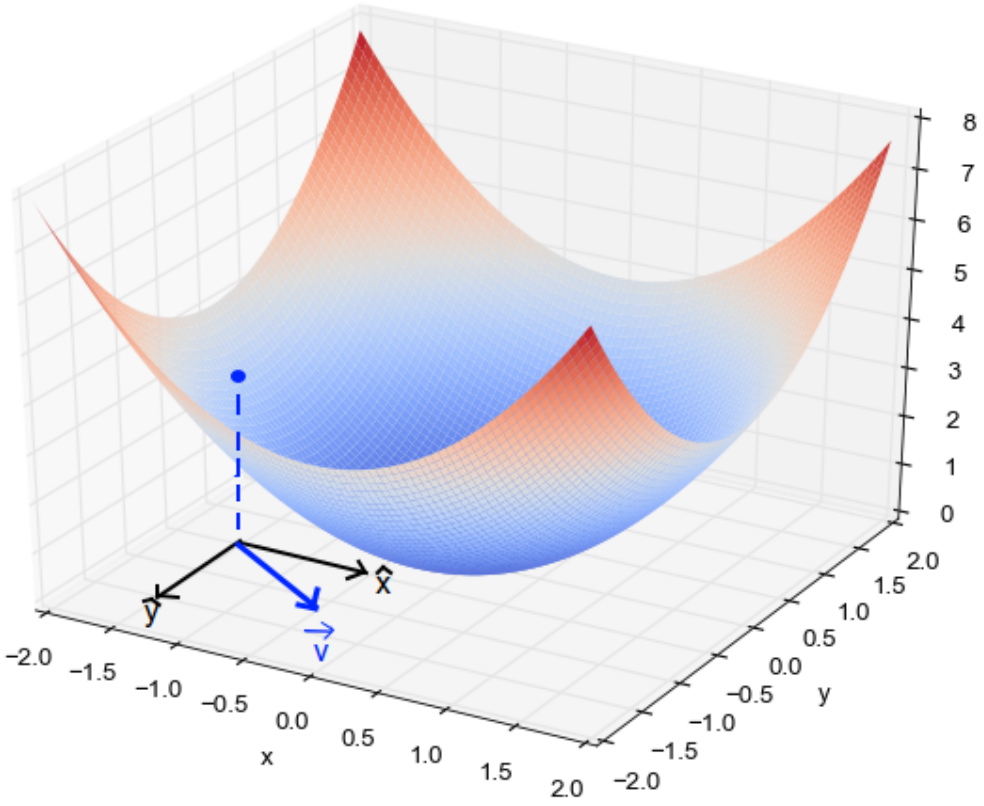
quedamos pues con esta modificación del algoritmo, eligiendo un valor de N que nos aporte un buen balance entre aleatoriedad y tiempo de entrenamiento (también que no sea demasiado grande para la memoria de la GPU disponible).

El Algoritmo

1… Introducimos un mini-lote de entrada con N muestras aleatorias provenientes de nuestro dataset de entrenamiento, previamente etiquetado (lo que significa que conocemos la salida real).
2… Después de los cálculos pertinentes en cada capa de la red, obtenemos como resultado las predicciones a su salida. A este paso se le conoce como forward propagation (de las entradas).
3… Evaluamos la función de coste (también llamada función de pérdida) para dicho mini-lote. Se trata de una función elegida previamente en base al tipo de problema concreto, para poder evaluar de la forma más adecuada la diferencia entre las predicciones de nuestra red y las salidas reales. El valor de esta función de coste es lo que se trata de minimizar en todo momento mediante el algoritmo, y hacia ello se orientan los siguientes pasos.

Problem Type	Output Type	Final Activation Function	Loss Function
Regression	Numerical value	Linear	Mean Squared Error (MSE)
Classification	Binary outcome	Sigmoid	Binary Cross Entropy
Classification	Single label, multiple classes	Softmax	Cross Entropy
Classification	Multiple labels, multiple classes	Sigmoid	Binary Cross Entropy

4… Calculamos el gradiente como la derivada multivariable de la función de coste con respecto a todos los parámetros de la red. Gráficamente sería la pendiente de la tangente a la función de coste en el punto donde nos encontramos (evaluando los pesos actuales). Matemáticamente es un vector que nos da la dirección y el sentido en que dicha función aumenta más rápido, por lo que deberíamos movernos en sentido contrario si lo que tratamos es de minimizarla.



¿Cómo saber lo que influye realmente la variación de un parámetro de la primera capa en el coste final si esa variación repercute en todas las neuronas de todas las capas sucesivas?

Menos mal que disponemos de un algoritmo conocido como back-propagation (o propagación hacia atrás). Dicho algoritmo consiste en comenzar calculando las derivadas parciales de la función de coste a la salida con respecto únicamente a los parámetros de la última capa (que no influyen sobre ningún otro parámetro de la red). No es un cálculo muy complicado gracias a la regla de la cadena.
Una vez obtenidas estas derivadas, pasamos a la capa anterior, y calculamos nuevamente las derivadas parciales de la función de coste, pero ahora con respecto a los parámetros de esta capa, algo que en parte ya tenemos resuelto precisamente por la regla de la cadena.
Y así seguiremos progresando hacia atrás, hasta llegar al inicio de la red.

5… Una vez obtenido el vector gradiente, actualizaremos los parámetros de la red restando a su valor actual el valor del gradiente correspondiente, multiplicado por una tasa de aprendizaje que nos permite ajustar la magnitud de nuestros pasos. El término de actualización se resta porque como ya hemos visto antes queremos avanzar en el sentido contrario al del gradiente, para que la función de coste disminuya. Según nos acerquemos al mínimo global, los pasos serán en teoría más pequeños porque la pendiente de la función de coste será menor, pero mejor dejemos de pensar en el ejemplo irreal en tres dimensiones: se suele optar igualmente por ir disminuyendo la tasa de aprendizaje con el tiempo.

Modelo de redes neuronales en python

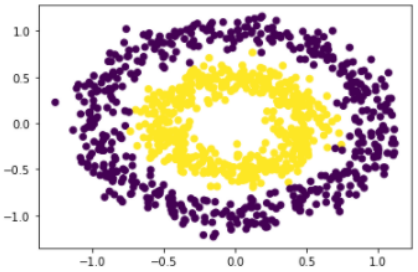
A continuación se desarrollará un ejemplo de implementación de una red neuronal artificial para resolver un problema no linealmente separable. Utilizaremos la función make_circles de sklearn para simular nuestro conjunto de datos. Esta función nos generará un conjunto de datos que, al graficarlos, forman dos círculos concéntricos. El círculo del centro corresponde a ejemplos de la clase A, mientras que el círculo de afuera corresponde a ejemplos de la clase B.
Comenzaremos importando las librerías que necesitamos

In [1]:
```
from sklearn.datasets import make_circles
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
import numpy as np
```

Y generaremos nuestro conjunto de datos artificial

In [2]:
```
x, y = make_circles(n_samples=1000, factor=0.5, noise=0.1, random_state=42)
x1 = x[:, 0].flatten()
x2 = x[:, 1].flatten()

plt.scatter(x1, x2, c=y)
plt.show()
```



Luego de eso, dividiremos nuestros datos en train y test para entrenar y testear el modelo

In [3]:
```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2)
```

Importaremos las librerias necesarias para trabajar con redes neuronales en python

In [4]:
```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.layers import Dense
from tensorflow.keras.models import Sequential
```

Utilizaremos el modelo Sequential para crear nuestra red neuronal. Dicha red, consta solamente de dos capas de 8 neuronas con función de activación relu mientras que la última capa tendrá una sola neurona con función de activación sigmoide

In [5]:
```
model = Sequential()
model.add(Dense(8, activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

Utilizaremos el optimizador adam, con la función de perdida binary_crossentropy puesto que nuestro problema de clasificación consta solamente de dos clases. También utilizaremos accuracy como métrica del modelo.

In [6]:
```
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

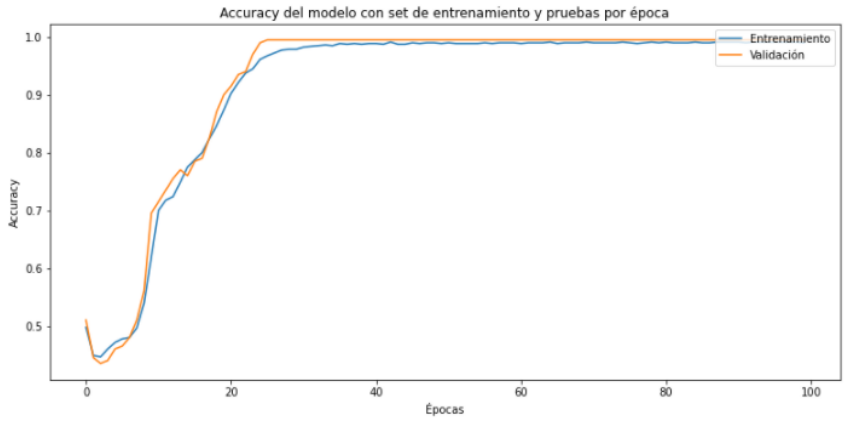
Entrenaremos al modelo durante 100 épocas utilizando un tamaño de batch de 32.

In [7]:
```
model.fit(x=x_train,
          y=y_train,
          batch_size=32,
          epochs=100,
          validation_data=(x_test, y_test))
```

```
Epoch 40/100
25/25 [==============================] - 0s 6ms/step - loss: 0.1062 - accuracy: 0.9936 - val_loss: 0.0908 - val_accuracy: 0.9
950
Epoch 41/100
25/25 [==============================] - 0s 6ms/step - loss: 0.1075 - accuracy: 0.9841 - val_loss: 0.0852 - val_accuracy: 0.9
950
Epoch 42/100
25/25 [==============================] - 0s 5ms/step - loss: 0.0981 - accuracy: 0.9900 - val_loss: 0.0807 - val_accuracy: 0.9
950
Epoch 43/100
25/25 [==============================] - 0s 5ms/step - loss: 0.1029 - accuracy: 0.9861 - val_loss: 0.0761 - val_accuracy: 0.9
950
Epoch 44/100
25/25 [==============================] - 0s 5ms/step - loss: 0.0928 - accuracy: 0.9873 - val_loss: 0.0718 - val_accuracy: 0.9
950
Epoch 45/100
25/25 [==============================] - 0s 4ms/step - loss: 0.0822 - accuracy: 0.9879 - val_loss: 0.0685 - val_accuracy: 0.9
950
Epoch 46/100
25/25 [==============================] - 0s 5ms/step - loss: 0.0823 - accuracy: 0.9897 - val_loss: 0.0650 - val_accuracy: 0.9
```

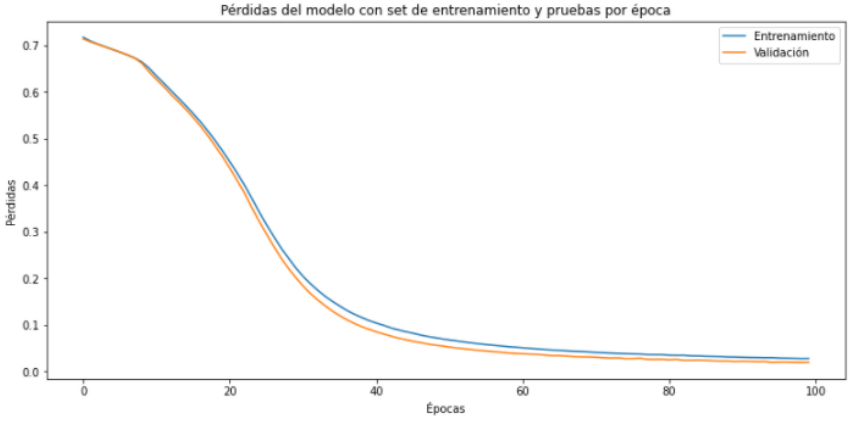
A continuación podemos observar las curvas de aprendizaje del modelo
Accuracy:

In [8]:
```
plt.figure(figsize=(13,6))
plt.plot(model.history.history['accuracy'])
plt.plot(model.history.history['val_accuracy'])
plt.title("Accuracy del modelo con set de entrenamiento y pruebas por época")
plt.ylabel('Accuracy')
plt.xlabel('Épocas')
plt.legend(['Entrenamiento', 'Validación'], loc='upper right')
plt.show()
```



Perdida:

In [9]:
```
plt.figure(figsize=(13,6))
plt.plot(model.history.history['loss'])
plt.plot(model.history.history['val_loss'])
plt.title("Pérdidas del modelo con set de entrenamiento y pruebas por época")
plt.ylabel('Pérdidas')
plt.xlabel('Épocas')
plt.legend(['Entrenamiento', 'Validación'], loc='upper right')
plt.show()
```



Métricas

Set de train:

```
In [10]: 1 model.evaluate(x_train, y_train)
```

```
25/25 [==============================] - 0s 2ms/step - loss: 0.0272 - accuracy: 0.9912
Out[10]: [0.027181386947631836, 0.9912499785423279]
```

Set de test:

```
In [11]: 1 model.evaluate(x_test, y_test)
```

```
7/7 [==============================] - 0s 1ms/step - loss: 0.0198 - accuracy: 0.9950
Out[11]: [0.019799835979938507, 0.9950000047683716]
```